Е.В. Шикин, А.В. Боресков

КОМПЬЮТЕРНАЯ ГРАФИКА

динамика,
реалистические
изображения

МОСКВА «ДИАЛОГ-МИФИ» 1997

УДК 681.3
Ш57

Шикин Е. В., Боресков А. В.
Ш57 Компьютерная графика. Динамика, реалистические изображения. -М.: ДИАЛОГ-МИФИ, 1996. - 288 с.

ISBN 5-86404-061-4

Книга знакомит с такими основными понятиями и методами компьютерной графики, как растровые алгоритмы, геометрические сплайны, методы удаления скрытых линий и поверхностей, закрашивание, трассировка лучей, излучательность. Она дает представление об основных направлениях компьютерной графики и позволяет освоить базовые приемы реализации ее алгоритмов на персональных компьютерах. В книге дается краткое описание основных возможностей графического пакета 3D Studio. Приведенные в книге программы могут быть использованы при решении широкого класса задач визуализации и анимации. Предлагаемая книга содержит материал, отобранный на основе многолетнего опыта чтения курса по компьютерной графике, а также анализа отзывов читатели на выпущенную ранее книгу "Начало компьютерной графики".

Книгу можно рассматривать как практическое руководство, т. к. она содержит примеры графических задач, которые способен выполнить, прочитавший книгу. Дополнительно можно приобрести дискету со сквозными иллюстрирующими примерами и рядом программ по визуализации и анимации.

Ш 2404000000.022 Без объявл.
Г70(03)-97

Учебно-справочное издание
Шикин Евгений Викторович
Боресков Алексей Викторович
Компьютерная графика.
Динамика, реалистические изображения

Редактор О. Л. Голубев
Корректор В. С. Кустов
Макет О. Л. Кузьминовой
Обложка Н. В. Дмитриевой
Лицензия ЛР N 070109 от 29.08.91. Подписано в печать 15.09.97.
Формат 60x84/16. Бум. офс. Печать офс. Гарнитура Таймс.
Усл. печ. л. 16.74. Уч.-изд. л. 8.9. Доп. тираж 2 000 экз. Заказ 362.
Акционерное общество "ДИАЛОГ-МИФИ" 115409, Москва, ул. Москворечье, 31, корп. 2
Подольская типография
142110, г. Подольск, Московская обл., ул. Кирова, 25

ISBN 5-86404-061-4

Светлой памяти наших отцов
Макарова Виктора Алексеевича и
Шикина Виктора Григорьевича
посвящаем

ПРЕДИСЛОВИЕ

Мы ясно знаем, что зрение одно из быстрейших действий, какие только существуют: в одной точке оно видит бесконечно много форм и тем не менее понимает сразу лишь один предмет.

Леонардо да Винчи

Немногим более года назад была выпущена книга "Начала компьютерной графики", в написании которой принимали участие оба автора и которая оказалась, по-существу, едва ли не единственной отечественной книгой по компьютерной графике за последние несколько лет.

Так получилось, что довольно быстро на выпущенную книжку было получено несколько десятков отзывов от заинтересованных читателей. Знакомство с этими отзывами подтвердило, в частности, наши собственные впечатления о том, что одной небольшой книги по компьютерной графике мало. В отзывах отмечалось, что определенная часть материала, включая описание цветовосприятия, методы закрашивания, сколь-либо подробный разговор об известных графических пакетах, осталась за ее пределами, а на некоторые важные направления, такие, как наиболее употребительные методы создания реалистических изображений и использование сплайнов в компьютерной графике, было отведено слишком мало места.

Высказанные пожелания приблизить изложение к пользователю и сопроводить сказанное поясняющими примерами вполне согласовывались с ощущениями авторов и явились одним из существенных толчков к подготовке и написанию предлагаемой книги.

В этой книге мы решили заметно (по сравнению с вышедшей) изменить структуру изложения, существенно переработать ранее представленный материал и добавить то, что, по нашему мнению, совершенно необходимо для создания на экране монитора динамических картинок и кадров, близких к реалистичным. Отдельную часть книги мы специально посвящаем описанию третьей версии пакета 3D Studio.

Кроме того, мы посчитали целесообразным сопроводить книгу дискетой, которая содержит не только тексты приведенных в книге программ, но и способна показать то, что сможет создать на экране персонального компьютера любой пытливый пользователь, прочитавший настоящую книгу.

АО "ДИАЛОГ-МИФИ" согласилось с нашими доводами в пользу издания новой книги по компьютерной графике и любезно предложило авторам воплотить на бумаге и на гибком носителе наши намерения, как ранее неосуществленные, так и новые. В свою очередь, авторы, высоко оценивая важность той просветительской работы, которую проводит издательский отдел АО "ДИАЛОГ-МИФИ" по выпуску учебно-ориентированной литературы в столь непростое для нашего образования время, выражают чувство искренней признательности всем тем, кто способствовал выходу в свет этой книги: Елене Константиновне Виноградовой, Олегу Александровичу Голубеву, Наталье Викторовне Дмитриевой и Оксане Алексеевне Кузьминовой.

Авторы благодарны руководителю IBS-дивизиона графических технологий Спартаку Петровичу Чеботареву и руководителю отдела прикладных проектов Рафаилу Ефимовичу Глуховскому за предоставленные материалы, а также директору НПП "Гарант-Сервис" Дмитрию Викторовичу Першееву за неизменно благожелательную поддержку.

Книга подготовлена при поддержке Российского Фонда фундаментальных исследований, грант 95-01-01471.

Боресков А. В. Шикин Е. В.

Июнь 1995 года

О читателе, на которого рассчитана книга

Желательно, чтобы читатель был знаком с языком программирования С++ и интересовался компьютерной графикой.

Требования к математической составляющей предварительных сведений ограничиваются знакомством с элементами векторной алгебры и математического анализа, а также начальными понятиями линейной алгебры и дифференциальной геометрии. Дополнительные сведения приведены в соответствующих местах книги в объеме, необходимом для понимания.

Об иллюстрациях

На иллюстрациях следует остановиться особо.

Значительная их часть носит вполне традиционный характер и состоит из простых и легко воспроизводимых рисунков, в разной степени поясняющих изложенное. В этих рисунках отражены (зачастую, в схематической форме) некоторые свойства описываемых понятий, явлений и фактов.

Другие иллюстрации представлены в виде, менее привычном читателю. Желание авторов книг, посвященных компьютерной графике, показать пользователю реализацию нижнего предела того, что открывает перед ним знакомство с материалом, помещенным на страницах книги, следует признать совершенно естественным. Например, в книгах по компьютерной графике, выпущенных за рубежом, результаты применения алгоритмов, представленных в книге, помешают на цветных вклейках. Эти вклейки занимают в книге заметное место и играют важную разъясняющую роль. Кроме того, они неизбежно привлекают внимание читателя и даже просто любопытствующего. Достаточно ясно, однако, что издание таких книг опирается на высокоразвитую полиграфическую базу и требует, заметим, немалых затрат. Вполне трезво оценивая имеющиеся возможности, авторы приняли решение при работе над книгой пойти по несколько иному пути. Они поместили в книгу готовые программы, которые при помощи персонального компьютера каждый пользователь сможет развернуть в разнообразные по содержанию картинки и получить тем самым новые иллюстрации к книге. Важно отметить, что некоторые из этих иллюстраций имеют достаточно выразительную динамику, чего практически невозможно добиться обычными средствами полиграфии.

Готовые программы повторены на дискете, прилагаемой к книге. Эта дискета содержит тексты и других программ, которые также можно развернуть на экране в набор иллюстраций при посредстве персонального компьютера.

О дискете

К книге дополнительно можно купить дискету, которая помимо исходных текстов всех рассмотренных в книге примеров (около 200 Кбайт) содержит еще целый ряд материалов: примеры, иллюстрации и программы, не вошедшие в книгу вследствие ограниченности ее объема.

В числе этих материалов несколько программ для работы с большинством распространенных SVGA-карт, набор дополнительных объектов и текстур для трассировки лучей и многое другое, могущее оказаться полезным читателю.

Распространением дискеты занимается АО "ДИАЛОГ-МИФИ" (тел. 320-43-77).

Авторы будут признательны читателям за предложения; и замечания, которые можно прислать по электронной почте непосредственно по адресу:

alex@garser.msk.su
shikin@cmc.msk.su

ВВЕДЕНИЕ

При обработке информации, связанной с изображением на мониторе, принято выделять три основных направления: распознавание образов, обработку изображений и машинную графику.

Основная задача распознавания образов состоит в преобразовании уже имеющегося изображения на формально понятный язык символов. Распознавание образов (изображений) есть совокупность методов, позволяющих получить описание изображения, поданного на вход, либо отнести заданное изображение к некоторому классу (так поступают, например, при сортировке почты или в медицинской диагностике, где путем анализа томограмм оценивают наличие отклонений от нормы). При этом рассматриваемое изображение часто преобразуется в более абстрактное описание - набор чисел, набор символов или граф. Одной из интересных задач распознавания образов является так называемая скелетизация объектов, при которой восстанавливается некая основа объекта, его "скелет".

Символически, распознавание изображений, или система технического зрения COMPUTER VISION, может быть описана так:

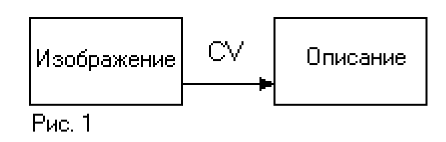


Рис. 1

- input - изображение;
- output - символ (текст) и его последующий анализ (рис. 1).

Обработка изображении рассматривает задачи, в которых и входные и выходные данные являются изображениями. Примерами обработки изображений могут служить: передача изображений вместе с устранением шумов и сжатием данных, переход от одного вида изображения (полутонового) к другому (каркасному), контактирование различных снимков, а также синтезирование имеющихся изображений в новые, например по набору поперечных сечений объекта построить продольные сечения или восстановить сам объект.

Тем самым система обработки изображений IMAGE PROCESSING имеет следующую структуру.

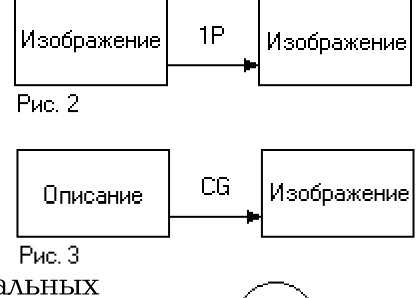


Рис. 2

Рис. 3

- input - изображение;
- output - изображение (преобразование изображений) (рис. 2).

Компьютерная (машинная) графика воспроизводит изображение в случае, когда исходной является информация не изобразительной природы. Например, визуализация экспериментальных данных в виде графиков, гистограмм или диаграмм, вывод информации на экран в компьютерных играх, синтез сцен для тренажеров.

А еще есть компьютерная живопись, компьютерная анимация и так далее, вплоть до виртуальной реальности.

Можно сказать, что компьютерная графика рисует, опираясь на формальные правила и имеющийся набор средств.

Символически систему компьютерной графики COMPUTER GRAPHICS можно представить следующим образом

- input - символьное описание;
- output - изображение (синтез изображений) (рис. 3).

Удобно нарисовать общую схему, вмещающую в себя описание и функции CV, 1P и CG, тем более, что резких границ между ними провести нельзя (рис. 4).

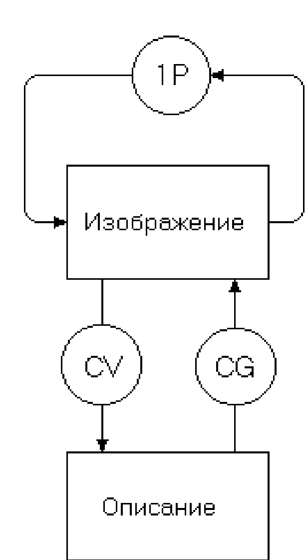


Рис. 4

Предметом настоящей книги является именно компьютерная графика.

Выделим некоторые ее направления (отметив, что это выделение достаточно условно):

1) иллюстративное, которое можно понимать расширительно, начиная с пояснений (визуализации) результатов эксперимента и кончая созданием рекламных роликов;
2) саморазвивающее - компьютерная графика должна обслуживать свои потребности, расширяя свои возможности и совершенствуя их;
3) исследовательское, в котором инструментарий компьютерной графики начинает играть роль, во многом подобную той, которую в свое время сыграл микроскоп.

Обычно изобретение инструмента и начало его массового и успешного применения разделены достаточно заметным промежутком времени. Имеются сведения, что прибор типа микроскопа был Построен около 1590 года. Но только в 1665 году (то есть через 75 лет) Гук впервые применил микроскоп в научных исследованиях, установив, в частности, клеточное строение растительных и животных тканей, а Левенгук при помощи микроскопа открыл (около 1675 года) микроорганизмы.

Микроскоп существенно раздвинул зрительные возможности человека, послужив мощным толчком к развитию изысканий в самых разных новых естественнонаучных направлениях. Сейчас время бежит значительно быстрее. И потому временной разрыв между первыми шагами по выводу изображения средствами компьютерной графики на экран и началом эффективного ее использования как мощного инструментария в научных исследованиях уже не столь велик. Впрочем, в полной мере возможности, которые несет в себе компьютерная графика, нам пока еще неясны.

Вывод изображения на экран персонального компьютера (сначала текста, формул, а затем и простейших рисунков) явился необходимым, но всего лишь первым шагом на пути становления компьютерной, или машинной, графики. Довольно стремительно пройдя иллюстративный отрезок пути своего развития, компьютерная графика сосредоточилась как бы на двух генеральных направлениях - придании изображению необходимой динамики и придании изображению необходимой реалистичности. К сожалению, пока большинство доступных персональных компьютеров не в состоянии достаточно удовлетворительно (то есть так, чтобы "радовался глаз") совмещать оба эти направления. Тем не менее по всему чувствуется, что указанное препятствие на пути совершенствования компьютерной графики и расширения ее возможностей вполне преодолимо. Надежду дает, в частности, тот постоянно нарастающий интерес, который естественно катализирует ведущиеся разработки.

Достижения компьютерной графики мы постоянно видим на экранах телевизоров в тех же рекламных заставках. Для большинства зрителей кажется удивительным, что все изображаемое существует в зашифрованном виде - в виде формул и текстов программ. Еще большее удивление, граничащее с почти нескрываемым недоверием к услышанному, вызывает у них количество людских и временных усилий, затрачиваемых на создание крохотного ролика. Реклама в данном случае выступает даже не столько как "двигатель торговли", сколько как мощный стимул к развитию все более совершенного, все более изощренного графического инструментария. И он существует в виде разнообразных графических пакетов - начиная от простейших графических редакторов и кончая программным обеспечением графических станций. Развитие компьютерной графики создало новый изобразительный инструмент, привлекший внимание дизайнеров, архитекторов, художников.

Перейдем к краткому описанию содержания книги. Она естественно разбивается на несколько частей.

В первой части описываются основные графические возможности персональных компьютеров и основные графические устройства.

Содержание второй части книги раскрывает подзаголовок в ее названии. В этой части последовательно, шаг за шагом показывается, как можно решить указанные выше основные задачи:

- придания изображению на экране необходимой динамики;
- построения на экране изображения сложной сцены, достаточно близкого к реальному.

Первые главы этой части более просты и фактически доступны любознательному школьнику.

Цель, которую поставили перед собой авторы, состоит в том, чтобы познакомить читателя с основными приемами, необходимыми для решения этих основных задач, и показать некоторые возможности этих приемов. Поэтому объекты, населяющие рассматриваемые в книге сцены, выбираются достаточно простыми. Что же касается эффективности в овладении этими приемами, то ее можно воспринимать уже как результат самостоятельной работы пользователя.

Собственно, создание на экране каркасного изображения простейших геометрических фигур обеспечивает графический модуль языка программирования высокого уровня. Описание простейших геометрических преобразований на плоскости и в пространстве позволяет перемещать построенные фигуры или проекции этих фигур в плоскости экрана. Введение динамики на экран уже па ранней стадии показывает принципиально иные изобразительные возможности персонального компьютера.

Удаление скрытых линий и частей поверхностей позволяет сделать изображаемые на экране объекты более привычными глазу пользователя. В том же направлении работают и методы закраски.

Заключительные главы второй части требуют уже достаточно высокой математической (в частности, геометрической) культуры и более уверенного владения программистским инструментарием.

Разумеется, далеко не все из окружающих нас объектов имеют многогранную форму. Довольно часто приходится иметь дело с гладкими двумерными объектами (без ребер и заострений). Конструирование сложных сглаженных объектов - кривых и поверхностей - основная задача, которая успешно решается при помощи геометрических сплайнов.

Завершает вторую часть рассмотрение двух мощных методов создания реалистических изображений - метода трассировки лучей и метода излучательности. Результаты их применения к сложным сценам,

вмещающим многие объекты, заметно приближаются к тому, что мы привыкли видеть вокруг.

Нельзя не сказать об объемах необходимых вычислений. Достаточно ясно, что чем более совершенным (реалистичным) выглядит изображение объекта, тем больших затрат требуют соответствующие расчеты. Последние два метода, а также создание динамических изображений сравнительно несложных сцен являются особенно трудоемкими. Поэтому на большинстве используемых персональных компьютеров динамика изображения сложной сцены и его реалистичность совмещаются довольно плохо. Тем не менее, во многих практически интересных случаях эти трудности удается преодолевать.

Третья часть посвящена описанию некоторых возможностей графического пакета 3D Studio.

Прилагаемая дискета рассматривается авторами как четвертая, заключительная часть книги.

ГРАФИЧЕСКИЕ ПРИМИТИВЫ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

На большинстве ЭВМ (включая и IBM PC/AT) принят растровый способ изображения графической информации - изображение представлено прямоугольной матрицей точек (пикселов), и каждый пиксел имеет свой цвет, выбираемый из заданного набора цветов - палитры. Для реализации этого подхода компьютер содержит в своем составе видеоадаптер, который, с одной стороны, хранит в своей памяти (ее принято называть видеопамятью) изображение (при этом на каждый пиксел изображения отводится фиксированное количество бит памяти), а с другой - обеспечивает регулярное (50-70 раз в секунду) отображение видеопамяти на экране монитора. Размер палитры определяется объемом видеопамяти, отводимой под один пиксел, и зависит от типа видеоадаптера.

Для ПЭВМ типа IBM PC/AT и PS/2 существует несколько различных типов видеоадаптеров, различающихся как своими возможностями, так и аппаратным устройством и принципами работы с ними. Основными видеоадаптерами для этих машин являются CGA, EGA, VGA и Hercules. Существует также большое количество адаптеров, совместимых с EGA/VGA, но предоставляющих по сравнению с ними ряд дополнительных возможностей.

Практически каждый видеоадаптер поддерживает несколько режимов работы, отличающихся друг от друга размерами матрицы пикселов (разрешением) и размером палитры (количеством цветов, которые можно одновременно отобразить на экране). Зачастую разные режимы даже одного адаптера имеют разную организацию видеопамяти и способы работы с ней. Более подробную информацию о работе с видеоадаптерами можно получить из следующей главы,

Однако большинство адаптеров строится по принципу совместимости с предыдущими. Так, адаптер EGA поддерживает все режимы адаптера CGA. Поэтому любая программа, рассчитанная на работу с адаптером CGA, будет также работать и с адаптером EGA, даже не замечая этого. При этом адаптер EGA поддерживает, конечно, еще ряд своих собственных режимов. Аналогично адаптер VGA поддерживает все режимы адаптера EGA.

Фактически любая графическая операция сводится к работе с отдельными пикселами - поставить точку заданного цвета и узнать цвет заданной точки. Однако большинство графических библиотек поддерживают работу и с более сложными объектами, поскольку работа только на уровне отдельно взятых пикселов была бы очень затруднительной для программиста и, к тому же, неэффективной.

Среди подобных объектов (представляющих собой объединения пикселов) можно выделить следующие основные группы:

- линейные изображения (растровые образы линий);
- сплошные объекты (растровые образы двумерных областей);
- шрифты;
- изображения (прямоугольные матрицы пикселов).

Как правило, каждый компилятор имеет свою графическую библиотеку, обеспечивающую работу с основными группами графических объектов. При этом требуется, чтобы подобная библиотека поддерживала работу с основными типами видеоадаптеров.

Существует несколько путей обеспечения этого.

Один из них заключается в написании версий библиотеки для всех основных типов адаптеров. Однако программист должен изначально знать, для какого конкретно видеоадаптера он пишет свою программу, и использовать соответствующую библиотеку. Полученная при этом программа уже не будет работать на других адаптерах, несовместимых с тем, для которого писалась программа. Поэтому вместо одной программы получается целый набор программ для разных видеоадаптеров. Принцип совместимости адаптеров выручает

здесь несильно: хотя программа, рассчитанная на адаптер CGA, и будет работать на VGA, но она не сможет полностью использовать все его возможности и будет работать с ним только как с CGA.

Можно включить в библиотеку версии процедур для всех основных типов адаптеров. Это обеспечит некоторую степень машинной независимости. Однако нельзя исключать случай наличия у пользователя программы какого-либо типа адаптера, не поддерживаемого библиотекой (например, SVGA). Но самым существенным недостатком такого подхода является слишком большой размер получаемого выполняемого файла, что уменьшает объем оперативной памяти, доступный пользователю.

Наиболее распространенным является использование драйверов устройств. При этом выделяется некоторый основной набор графических операций так, что все остальные операции можно реализовать, используя только операции основного набора. Привязка к видеоадаптеру заключается именно в реализации этих основных (базисных) операций. Для каждого адаптера пишется так называемый драйвер - небольшая программа со стандартным интерфейсом, реализующая все эти операции для данного адаптера и помещаемая в отдельный файл. Библиотека в начале своей работы определяет тип

имеющегося видеоадаптера и загружает соответствующий драйвер в память. Таким образом достигается почти полная машинная независимость написанных программ.

Рассмотрим работу одной из наиболее популярных графических библиотек - библиотеки компилятора Borland C++. Для использования этой библиотеки необходимо сначала подключить ее при помощи команды меню Options/Linker/Libraries.

Рассмотрим основные группы операций.

Инициализация и завершение работы с библиотекой

Для инициализации библиотеки служит функция

```
void far initgraph (int far *driver, int far *mode, char far *path);
```

Первый параметр задает библиотеке тип адаптера, с которым будет вестись работа. В соответствии с этим параметром будет загружен драйвер указанного видеоадаптера и произведена инициализация всей библиотеки. Определен ряд констант, задающих набор стандартных драйверов: CGA, EGA, VGA, DETECT и другие.

Значение DETECT сообщает библиотеке о том, что тип имеющегося видеоадаптера надо определить ей самой и выбрать для него режим наибольшего разрешения.

Второй параметр - mode - определяет режим.

Параметр	*Режим*
CGAC0, CGAC1, CGAC2, CGAC3	320 на 200 точек на 4 цвета
CGAHI	640 на 200 точек на 2 цвета
EGALO	640 на 200 точек на 16 цветов
EGAHI	640 на 350 точек на 16 цветов
VGALO	640 на 200 точек на 16 цветов
VGAMED	640 на 350 точек на 16 цветов
VGAHI	640 на 480 точек на 16 цветов

Если в качестве первого параметра было взято значение DETECT, то параметр mode не используется.

В качестве третьего параметра выступает имя каталога, где находится драйвер адаптера - файл типа BGI (Borland's Graphics Interface):

- CGA.BGI - драйвер адаптера CGA;
- EGAVGA.BGI - драйвер адаптеров EGA и VGA;
- HERC.BGI - драйвер адаптера Hercules.

Функция graphresult возвращает код завершения предыдущей графической операции

```
int far graphresult ( void );
```

Успешному выполнению соответствует значение grOk.

Для окончания работы с библиотекой необходимо вызвать функцию closegraph:

```
void far closegraph ( void );
```

```
// File example1.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

main ()
{
   int   driver = DETECT;
   int   mode;
   int   res;

   initgraph ( &driver, &mode, "" );
   if ( ( res = graphresult () ) != grOk )
   {
         printf("\nGraphics error: %s\n", grapherrormsg ( res) );
         exit ( 1 );
   }

   line ( 0, 0, 0, getmaxy () );
   line ( 0, getmaxy (), getmaxx (), getmaxy () );
   line ( getmaxx (), getmaxy (), getmaxx (), 0 );
   line ( getmaxx (), 0, 0, 0 );

   getch ();
   closegraph ();
}
```

Программа переходит в графический режим и рисует по краям экрана прямоугольник. В случае ошибки выдается стандартное диагностическое сообщение. После инициализации библиотеки адаптер переходит в соответствующий режим, экран очищается и на нем устанавливается следующая координатная система (рис. 1). Начальная точка с координатами (0, 0) располагается в левом верхнем углу экрана.

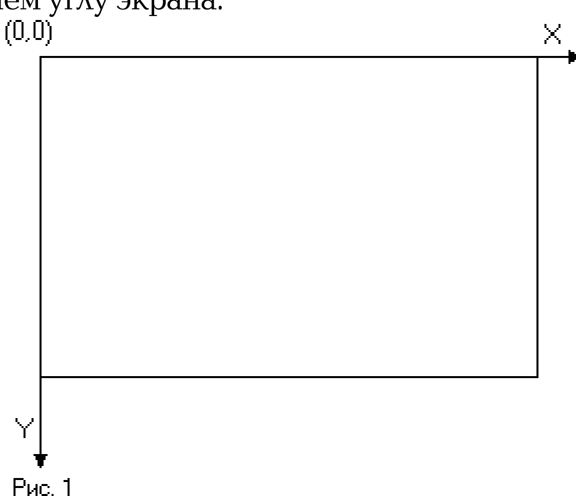


Рис. 1

Узнать максимальные значения X и Y координат пиксела можно, используя функции getmaxx и getmaxy:

```
int far getmaxx ( void );
int far getmaxy ( void );
```

Узнать, какой именно режим в действительности установлен, можно при помощи функции getgraphmode:

```
int far getgraphmode ( void );
```

Для очистки экрана удобно использовать функцию clearviewport:

```
void far clearviewport ( void );
```

Работа с отдельными точками

Функция putpixel ставит пиксел заданного цвета Color в точке с координатами (x, y):

```
void far putpixel ( int x, int y, int Color );
```

Функция getpixel возвращает цвет пиксела с координатами (x, y):

```
unsigned far getpixel ( int x, int y );
```

Рисование линейных объектов

При рисовании линейных объектов основным инструментом является перо, которым эти объекты рисуются. Перо имеет следующие характеристики:

- цвет (по умолчанию белый);
- толщина (по умолчанию 1);
- шаблон (по умолчанию сплошной).

Шаблон служит, для рисования пунктирных и штрихпунктирных линий. Для установки параметров пера используются следующие функции выбора.

Процедура setcolor устанавливает цвет пера:

```
void far setcolor ( int Color );
```

Функция setlinestyle определяет остальные параметры пера:

```
void far setlinestyle ( int Style, unsigned Pattern, int Thickness );
```

Первый параметр задает шаблон линии. Обычно в качестве этого параметра выступает один из предопределенных шаблонов: SOLID_LINE, DOTTED_LINE, CENTER_LINE, DASHED_LINE, USERBIT_LINE и другие. Значение USERBIT_LINE означает, что шаблон задается (пользователем) вторым параметром. Шаблон определяется 8 битами, где значение бита 1 означает, что в соответствующем месте будет поставлена точка, а значение 0 - что точка ставиться не будет.

Третий параметр задает толщину линии в пикселах. Возможные значения параметра - NORM_WIDTH и THICK_WIDTH (1 и 3).

При помощи пера можно рисовать ряд линейных объектов -прямолинейные отрезки, дуги окружностей и эллипсов, ломаные.

Рисование прямолинейных отрезков

Функция line рисует отрезок, соединяющий точки (x_1, y_1) и (x_2,y_2):

```
void far line ( int x1, int y1, int x2, int y2 );
```

Рисование окружностей

Функция circle рисует окружность радиуса r с центром в точке (x, y):

```
void far circle ( int x, int y, int r );
```

Рисование дуг эллипса

Функции arc и ellipse рисуют дуги окружности (с центром в точке (x, y) и радиусом r) и эллипса (с центром (x, y), полуосями rx и ry, параллельными координатным осям), начиная с угла StartAngle и заканчивая углом EndAngle.

Углы задаются в градусах в направлении против часовой стрелки (рис. 2):

```
void far arc (int x, int y, int StartAngle, int EndAngle, int r);
void far ellipse (int x, int y, int StartAngle, int EndAngle, int rx, int ry);
```

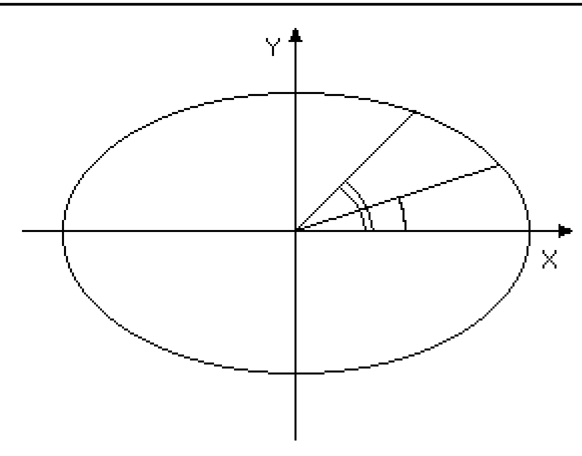


Рис. 2

Рисование сплошных объектов

Закрашивание объектов

С понятием закрашивания тесно связано понятие кисти. Кисть определяется цветом и шаблоном - матрицей 8 на 8 точек (бит), где бит, равный 1, означает, что нужно ставить точку цвета кисти, а 0 -что нужно ставить черную точку (цвета 0).

Для задания кисти используются следующие функции:

```
void far setfillstyle ( int Pattern, int Color );
void far setfillpattern (char far * Pattern, int Color );
```

Функция setfillstyle служит для задания кисти. Параметр Style определяет шаблон кисти либо как один из стандартных (EMPTY_FILL, SOLID_FILL, LINE_FILL, LTSLASH_FILL), либо как шаблон, задаваемый пользователем (USER_FILL). Пользовательский шаблон устанавливает процедура setfillpattem, первый параметр в которой и задает шаблон - матрицу 8 на 8 бит, собранных по горизонтали в байты. По умолчанию используется сплошная кисть (SOLID_FILL) белого цвета.

Процедура bar закрашивает выбранной кистью прямоугольник с левым верхним углом (x_1, y_1) и правым нижним углом (x_2, y_2)

```
void far bar ( int x1, int y1, int x2, int y2 );
```

Функция fillellipse закрашивает сектор эллипса:

```
void far fillellipse (int x, int y, int StartAngle, int EndAngle, int rx, int ry);
```

Функция floodfill служит для закраски связной области, ограниченной линией цвета BorderColor и содержащей точку (x, y) внутри себя:

```
void far floodfill ( int x, int y, int BorderColor );
```

Функция fillpoly осуществляет закраску многоугольника, заданного массивом значений x- и y-координат:

```
void far fillpoly ( int numpoints, int far * points );
```

Работа с изображениями

Библиотека поддерживает также возможность запоминания прямоугольного фрагмента изображения в обычной (оперативной) памяти и вывода его на экран. Это может использоваться для запоминания изображения в файл, создания мультипликации и т. п.

Объем памяти, требуемый для запоминания фрагмента изображения, в байтах можно получить при помощи функции imagesize:

```
unsigned far imagesize (int x1, int y1, int x2, int y2 );
```

Для запоминания изображения служит процедура getimage:

```
void far getimage (int x1, int y1, int x2, int y2, void far * Image);
```

При этом прямоугольный фрагмент, определяемый точками (x_1,y_1 и (x_2,y_2), записывается в область памяти, задаваемую последним параметром - Image.

Для вывода изображения служит процедура putimage:

```
void far putimage (int x, int y, void far * Image, int op);
```

Хранящееся в памяти изображение, которое задается параметром Image, выводится на экран так, чтобы точка (x, y) была верхним левым углом изображения. Последний параметр определяет способ наложения выводимого изображения на уже имеющееся на экране (см. функцию setwritemode). Поскольку значение (цвет) каждого пиксела представлено фиксированным количеством бит, то в качестве возможных вариантов наложения выступают побитовые логические операции. Возможные значения для параметра op приведены ниже:

- COPY_PUT - происходит простой вывод (замещение);
- NOT_PUT - происходит вывод инверсного изображения;
- OR_PUT - используется побитовая операция ИЛИ;
- XOR_PUT - используется побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ;
- AND_PUT - используется побитовая операция И.

```
 // get/putimage example
unsigned ImageSize = imagesize ( x1, y1, x2, y2 );
...
void     * Image = malloc ( ImageSize );
if ( Image != NULL )
  getimage ( x1, y1, x2, y2, Image );
...
if ( Image != NULL )
{

  putimage ( x, y, Image, COPY_PUT );
   free ( Image );
}
```

В этой программе происходит динамическое выделение под заданный фрагмент изображения на экране требуемого объема памяти. Этот фрагмент запоминается в отведенную память. Далее сохраненное изображение выводится на новое место (в вершину левого верхнего угла - (x, y)) и отведенная под изображение память освобождается.

Работа со шрифтами

Под шрифтом обычно понимается набор изображений символов. Шрифты могут различаться по организации (растровые и векторные), по размеру, по направлению вывода и по ряду других параметров. Шрифт может быть фиксированным (размеры всех символов совпадают) или пропорциональным (высоты символов совпадают, но они могут иметь разную ширину).

Для выбора шрифта и его параметров служит функция settextstyle:

```
void far settextstyle (int Font, int Direction, int Size );
```

Здесь параметр Font задает идентификатор одного из шрифтов:

- DEFAULT_FONT - стандартный растровый шрифт размером 8 на 8 точек, находящийся в ПЗУ видеоадаптера;
- TRIPLEX_FONT, GOTHIC_FONT, SANS_SERIF_FONT, SMALL_ FONT - стандартные пропорциональные векторные шрифты, входящие в комплект Borland C++ (шрифты хранятся в файлах типа CHR и по этой команде подгружаются в оперативную память; файлы должны находиться в том же каталоге, что и драйверы устройств). Параметр Direction задает направление вывода:
- HORIZ_DIR - вывод по горизонтали;
- VERT_DIR - вывод по вертикали.

Параметр Size задает, во сколько раз нужно увеличить шрифт перед выводом на экран. Допустимые значения 1, 2, ..., 10.

При желании можно использовать любые шрифты в формате CHR. Для этого надо сначала загрузить шрифт при помощи функции:

```
int far installuserfont ( char far * FontFileNarme );
```

а затем возвращенное функцией значение передать settextstyle в качестве идентификатора шрифта:

```
int MyFont = installuserfont ( "MYFONT.CHR" );
settextstyle ( MyFont, HORIZ_DIR, 5 );
```

Для вывода текста служит функция outtextxy:

```
void far outtextxy ( int x, int y, char far * text );
```

При этом строка text выводится так, что точка (x, y) оказывается вершиной левого верхнего угла первого символа.

Для определения размера, который займет на экране строка текста при выводе текущим шрифтом, используются функции, возвращающие ширину и высоту в пикселах строки текста:

```
int far textwidth ( char far * text ); int far textheight (char far * text );
```

Понятие режима (способа) вывода

При выводе изображения на экран обычно происходит замещение пиксела, ранее находившегося на этом месте, на новый. Можно, однако, установить такой режим, что в видеопамять будет записываться результат наложения ранее имевшегося значения на выводимое. Поскольку каждый пиксел представлен фиксированным количеством бит, то естественно, что в качестве такого наложения выступают

побитовые операции. Для установки используемой операции служит процедура setwritemode:

```
void far setwritemode ( int Mode);
```

Параметр Mode задает способ наложения и может принимать одно из следующих значений:

- COPY_PUT- происходит простой вывод (замещение);
- XOR_PUT - используется побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ.

Режим XOR_PUT удобен тем, что повторный вывод одного и того же изображения на то же место уничтожает результат первого вывода, восстанавливая изображение, которое было на экране до этого.

Замечание.

Не все функции графической библиотеки поддерживают использование режимов вывода; например, функции закраски игнорируют установленный режим наложения (вывода). Кроме того, некоторые функции могут не совсем корректно работать в режиме XOR___PUT.

Понятие окна (порта вывода)

При желании пользователь может создать на экране окно - своего рода маленький экран со своей локальной системой координат. Для этого служит функция setviewport:

```
void far setviewport (int x1, int y1, int x2, int y2, int Clip);
```

Эта функция устанавливает окно с глобальными координатами (x_1,y_1)-(x_2,y_2)- При этом локальная система координат вводится так, что точке с координатами (0, 0) соответствует точка с глобальными координатами (x_1,y_1). Это означает, что локальные координаты отличаются от глобальных координат лишь сдвигом на (x_1,y_1), причем все процедуры рисования (кроме SetViewPort) работают всегда с локальными координатами. Параметр Clip определяет, нужно ли проводить отсечение изображения, не помещающегося внутрь окна, или нет.

За*мечание*

Отсечение ряда объектов проводится не совсем корректно, так, функция outtextxy производит отсечение не на уровне пикселов, а по символам.

Понятие палитры

Адаптер EGA и все совместимые с ним адаптеры предоставляют дополнительные возможности по управлению цветом. Наиболее распространенной схемой представления цветов для видеоустройств является так называемое RGB-представление, в котором любой цвет представляется как сумма трех основных цветов - красного (Red), зеленого (Green) и синего (Blue) с заданными интенсивностями. Вес возможное пространство цветов представляет из себя единичный куб, и каждый цвет определяется тройкой чисел (r, g, b).

Например желтый цвет задается как (1, 1, 0), а малиновый - как (1, 0, 1). Белому цвету соответствует набор (1, 1, 1), а черному - (0, 0, 0).

Обычно под хранение каждой из компонент цвета отводится фиксированное количество n бит памяти. Поэтому считается, что допустимый диапазон значений для компонент цвета не [0, 1], а $[0, 2^n-1]$.

Практически любой видеоадаптер способен отобразить значительно большее количество цветов, чем определяется количеством бит, отводимых в видеопамяти под один пиксел. Для использования этой возможности вводится понятие палитры.

Палитра - это массив, в котором каждому возможному значению пиксела сопоставляется значение цвета (r, g, b), выводимое на экран. Размер палитры и ее организация зависят от типа используемого видеоадаптера.

Наиболее простой является организация палитры на EGA-адаптере. Под каждый из 16 возможных логических цветов (значений пиксела) отводится 6 бит, по 2 бита на каждую цветовую компоненту. При этом цвет в палитре задается байтом следующего вида: 00rgbRGB, где r, g, b, R, G, B могут принимать значение 0 или 1.

Используя функцию setpalette -

```
void far setpalette ( int Color, int ColorValue );
```

можно для любого из 16 логических цветов задать любой из 64 возможных физических цветов.

Функция getpalette -

```
void far getpalette ( struct palettetype far * palette );
```

служит для получения текущей палитры, которая возвращается в виде следующей структуры:

```
struct palettetype {
unsigned char size;
signed char colors [MAXCOLORS+1];
};
```

Следующая программа демонстрирует использование палитры для получения четырех оттенков красного цвета.

```
// File example2.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

main ()
{
   int   driver = DETECT;
   int   mode;
   int   res;
   int   i;
```

```
    initgraph ( &driver, &mode, "" );
    if ( ( res = graphresult () ) != grOk )
    {
          printf("\nGraphics error: %s\n", grapherrormsg ( res) );
          exit ( 1 );
    }

    setpalette ( 0, 0  );
    setpalette ( 1, 32 );
    setpalette ( 2, 4  );
    setpalette ( 3, 36 );

    bar ( 0, 0, getmaxx (), getmaxy () );

    for ( i = 0; i < 4; i++ )
    {
          setfillstyle ( SOLID_FILL, i );
          bar ( 120 + i*100, 75, 219 + i*100, 274 );
    }

    getch ();
    closegraph ();
}
```

Реализация палитры для 16-цветных режимов адаптера VGA намного сложнее. Помимо поддержки палитры адаптера EGA, видеоадаптер дополнительно содержит 256 специальных DAC-регистров, где для каждого цвета хранится его 18-битовое представление (по 6 бит на каждую компоненту). При этом исходному логическому номеру цвета с использованием 6-битовых регистров палитры EGA сопоставляется, как и раньше, значение от 0 до 63, но оно уже является не RGB-разложением цвета, а номером DAC-регистра, содержащего физический цвет.

Для установки значений DAC-регистров служит функция Setrgbpalette:

```
void far setrgbpalette ( int Color, int Red, int Green, int Blue );
```

Следующий пример переопределяет все 16 цветов адаптера VGA в 16 оттенков серого цвета.

```
// File example3.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

main ()
{
    int         driver = VGA;
    int         mode   = VGAHI;
    int         res;
    palettetype     pal;

    initgraph ( &driver, &mode, "" );

    if ( ( res = graphresult () ) != grOk )
    {
          printf("\nGraphics error: %s\n", grapherrormsg ( res) );
          exit ( 1 );
    }

    getpalette ( &pal );
    for ( int i = 0; i < pal.size; i++ )
    {
          setrgbpalette ( pal.colors [i], (63*i)/15, (63*i)/15, (63*i)/15 );
          setfillstyle  ( SOLID_FILL, i );
          bar ( i*40, 100, 39 + i*40, 379 );
```

```
    }

    getch ();
    closegraph ();
}
```

Для 256-цветных режимов адаптера VGA значение пиксела непосредственно используется для индексации массива DAC-регистров.

Понятие видеостраниц и работа с ними

Для большинства режимов (например, для EGAHI) объем видео памяти, необходимый для хранения всего изображения (экрана) составляет менее половины имеющейся видеопамяти (256 Кбайт для EGA и VGA). В этом случае вся видеопамять делится на равные части (их количество обычно является степенью двух), называемые страницами, так, что для хранения всего изображения достаточно одной и страниц. Для режима EGAHI видеопамять делится на две страницы 0-ю (адрес 0xA000:0) и 1-ю (адрес 0xA000: 0x8000).

Видеоадаптер отображает на экран только одну из имеющихся у него страниц. Эта страница называется видимой и устанавливается следующей процедурой:

```
void far setvisualpage ( int Page );
```

где Page - номер той страницы, которая станет видимой на экране после вызова этой процедуры.

Графическая библиотека также может осуществлять работу с любой из имеющихся страниц. Страница, с которой работает библиотека, называется активной. Активная страница устанавливается процедурой setactivepage:

```
void far setactivepage ( int Page );
```

где Page - номер страницы, с которой работает библиотека и на которую происходит весь вывод.

Использование видеостраниц играет очень большую роль при мультипликации.

Реализация мультипликации на ПЭВМ заключается в последовательном рисовании на экране очередного кадра. При традиционном способе работы (кадр рисуется, экран очищается, рисуется следующий кадр) постоянные очистки экрана и построение нового изображения па чистом экране создают нежелательный эффект мерцания.

Для устранения этого эффекта очень удобно использовать страницы видеопамяти. При этом, пока на видимой странице пользователь видит один кадр, активная, но невидимая страница очищается и на ней рисуется новый кадр. Как только кадр готов, активная и видимая страницы меняются местами и пользователь вместо старого кадра сразу видит новый.

```
// File example4.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

int      xc = 450;          // center of circle
int      yc = 100;
int      vx = 7;                   // velocity
int      vy = 5;
int      r = 20;                   // radius

void     DrawFrame ( int n )
{
   if ( ( xc += vx ) >= getmaxx () - r || xc < r )
   {
         xc -= vx;
         vx  = -vx;
```

```
    }

    if ( ( yc += vy ) >= getmaxy () - r || yc < r )
    {
          yc -= vy;
          vy  = -vy;
    }

    circle ( xc, yc, r );
}

main ()
{
    int   driver = EGA;
    int   mode   = EGAHI;
    int   res;

    initgraph ( &driver, &mode, "" );
    if ( ( res = graphresult () ) != grOk )
    {
          printf("\nGraphics error: %s\n", grapherrormsg ( res) );
          exit ( 1 );
    }

    DrawFrame ( 0 );
    setactivepage ( 1 );

    for ( int frame = 1;; frame++ )
    {
          clearviewport ();
          DrawFrame ( frame );
          setactivepage ( frame % 2 );
          setvisualpage ( 1 - ( frame % 2 ) );

          if ( kbhit () )
                break;
    }

    getch ();
    closegraph ();
}
```

Замечание

Не все режимы поддерживают работу с несколькими страницами, например VGAHI поддерживает работу только с одной страницей.

Подключение нестандартных драйверов устройств

Иногда возникает необходимость использовать нестандартные драйверы устройств. Это может возникнуть, например, в случае, если вы хотите работать с режимом адаптера VGA разрешением 320 на 200 точек при количестве цветов 256 или режимами адаптера SVGA. Эти режимы не поддерживаются стандартными драйверами, входящими в комплект Borland C++. Однако существует ряд специальных драйверов, предназначенных для работы с этими режимами. Приведем пример программы, подключающей драйвер для работы с 256-цветным режимом высокого разрешения для VESA-совместимого адаптера SVGA и устанавливающей палитру из 64 оттенков желтого цвета.

```
// File example5.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>
```

```
int      huge  MyDetect ( void )
{
   return 2;
}

main ()
{
   int   driver = DETECT;
   int   mode;
   int   res;

   installuserdriver ( "VESA", MyDetect );
   initgraph ( &driver, &mode, "" );

   if ( ( res = graphresult () ) != grOk )
   {
         printf("\nGraphics error: %s\n", grapherrormsg ( res) );
         exit ( 1 );
   }

   for ( int i = 0; i < 64; i++ )
   {
         setrgbpalette ( i, i, i, 0 );
         setfillstyle ( SOLID_FILL, i );
         bar ( i*10, 0, 9 + i*10, getmaxy () );
   }

   getch ();
   closegraph ();
}
```

При этом последним параметром для функции installuserdriver является функция, определяющая наличие соответствующей карты и возвращающей рекомендуемый режим.

Существует еще один способ подключения нестандартного драйвера устройства, когда вместо адреса проверяющей функции передается NULL, а возвращенное функцией installuserdriver значение используется в качестве первого параметра для функции initgraph.

```
// File example6.cpp
int driver;
int mode;
int res;
if ((driver = installuserdriver ("VESA", NULL )) == grError )
{
  printf ("\nCannot load extended driver" );
exit (1);
}
initgraph (&driver, &mode, "");
```

РАБОТА С ОСНОВНЫМИ ГРАФИЧЕСКИМИ УСТРОЙСТВАМИ

Несмотря на наличие различных графических библиотек (например, в составе компилятора Borland C++), часто возникает необходимость прямой работы с тем или иным графическим устройством. Это может быть связано как с тем, что библиотека не поддерживает соответствующее устройство (например, мышь или принтер), так и с тем, что работа с данным устройством организована недостаточно эффективно и всех его возможностей не использует.

Рассмотрим основные приемы работы с некоторыми устройствами.

Мышь

Наиболее распространенным устройством ввода графической информации в ПЭВМ является мышь. При перемещении мыши и/или нажатии/отпускании кнопок мышь передает информацию в компьютер о своих параметрах (величине перемещения и статусе кнопок). Существует много различных типов устройства типа мышь, отличающихся как по принципу работы (механическая, оптомеханическая и оптическая), так и по способу общения (протоколу) с ПЭВМ. Для достижения некоторой унификации каждая мышь поставляется обычно вместе со своим драйвером - специальной программой, понимающей данный конкретный тип мыши и предоставляющей некоторый (почти универсальный) интерфейс прикладным программам. При этом вся работа с мышью происходит через драйвер, который отслеживает перемещения мыши, нажатие и отпускание кнопок мыши и обеспечивает работу с курсором мыши - специальным маркером на экране (обычно в виде стрелки), дублирующим все передвижения мыши и позволяющим пользователю указывать мышью на те или иные объекты на экране.

Работа с мышью реализуется через механизм прерываний. Прикладная программа осуществляет прерывание 33h, передавая в регистрах необходимые параметры, и в регистрах же получает значения, возвращенные драйвером мыши.

Приводим набор функций для работы с мышью в соответствии со стандартом фирмы Microsoft. Ниже приведены используемые файлы Mouse.h и Mouse.cpp.

```
// File mouse.h
#ifndef  __MOUSE__
#define  __MOUSE__

#define  MOUSE_MOVE_MASK         0x01
#define  MOUSE_LBUTTON_PRESS     0x02
#define  MOUSE_LBUTTON_RELEASE   0x04
#define  MOUSE_RBUTTON_PRESS     0x08
#define  MOUSE_RBUTTON_RELEASE   0x10
#define  MOUSE_MBUTTON_PRESS     0x20
#define  MOUSE_MBUTTON_RELEASE   0x40
#define  MOUSE_ALL_EVENTS  0x7F

struct   MouseState
{
   int   x, y;
   int   Buttons;
};

struct   CursorShape
{
   unsigned AndMask [16];
   unsigned XorMask [16];
   int    HotX, HotY;
};
```

```
typedef  void  ( *MouseHandler )( int, int, int, int );

int      ResetMouse ();
void     ShowMouseCursor ();
void     HideMouseCursor ();
void     ReadMouseState ( MouseState& );
void     MoveMouseCursor ( int, int );
void     SetMouseHorzRange ( int, int );
void     SetmouseVertRange ( int, int );
void     SetMouseShape ( CursorShape& );
void    SetHideRange ( int, int, int, int );
void     SetMouseHandler ( MouseHandler, int = MOUSE_ALL_EVENTS );
void     RemoveMouseHandler ();

#define  ClearHideRange () ShowMouseCursor ()
#endif
```

```
// File mouse.cpp
#include <alloc.h>
#include "Mouse.h"

#pragma inline

static   MouseHandlerCurHandler = NULL;

int      ResetMouse ()
{
   asm {
         xor   ax, ax
         int   33h
   }

   return _AX == 0xFFFF;
}

void     ShowMouseCursor ()
{
   asm {
         mov   ax, 1
         int   33h
   }
}

void     HideMouseCursor ()
{
   asm {
         mov   ax, 2
         int   33h
   }
}

void     ReadMouseState ( MouseState& s )
{
   asm {
         mov   ax, 3
         int   33h
   }

#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
   asm {
         push  es
         push  di
         les   di, dword ptr s
         mov   es:[di  ], cx
         mov   es:[di+2], dx
```

```
        mov   es:[di+4], bx
        pop   di
        pop   es
   }
#else
   asm {
        push  di
        mov   di, word ptr s
        mov   [di  ], cx
        mov   [di+2], dx
        mov   [di+4], bx
        pop   di
   }
#endif
}

void     MoveMouseCursor ( int x, int y )
{
   asm {
        mov   ax, 4
        mov   cx, x
        mov   dx, y
        int   33h
   }
}

void     SetHorzMouseRange ( int xmin, int xmax )
{
   asm {
        mov   ax, 7
        mov   cx, xmin
        mov   dx, xmax
        int   33h
   }
}

void     SetVertMouseRange ( int ymin, int ymax )
{
   asm {
        mov   ax, 8
        mov   cx, ymin
        mov   dx, ymax
        int   33h
   }
}

void     SetMouseShape ( CursorShape& c )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
   asm {
        push  es
        push  di
        les   di, dword ptr c
        mov   bx, es:[di+16]
        mov   cx, es:[di+18]
        mov   dx, di
        mov   ax, 9
        int   33h
        pop   di
        pop   es
   }
#else
   asm {
        push  di
        mov   di, word ptr c
        mov   bx, [di+16]
```

```
            mov   cx, [di+18]
            mov   dx, di
            mov   ax, 9
            int   33h
            pop   di
      }
   #endif
   }

   void    SetHideRange ( int x1, int y1, int x2, int y2 )
   {
      asm {
            push  si
            push  di
            mov   ax, 10h
            mov   cx, x1
            mov   dx, y1
            mov   si, x2
            mov   di, y2
            int   33h
            pop   di
            pop   si
      }
   }

   static void far MouseStub ()
   {
      asm {
            push  ds    // preserve ds
            push  ax    // preserve ax
            mov   ax, seg CurHandler
            mov   ds, ax
            pop   ax    // restore ax
            push  dx    // y
            push  cx    // x
            push  bx    // button state
            push  ax    // event mask
            call  CurHandler
            add   sp, 8 // clear stack
            pop   ds
      }
   }

   void     SetMouseHandler ( MouseHandler h, int mask )
   {
      void  far * p = MouseStub;

      CurHandler = h;

      asm {
            push  es
            mov   ax, 0Ch
            mov   cx, mask
            les   dx, p
            int   33h
            pop   es
      }
   }

   void     RemoveMouseHandler ()
   {
      CurHandler = NULL;

      asm {
            mov   ax, 0Ch
            mov   cx, 0
```

```
            int   33h
        }
    }
```

Инициализация и проверка наличия мыши

Функция Reset Mouse производит инициализацию мыши и возвращает ненулевое значение, если мышь обнаружена.

Высветить на экране курсор мыши

Функция ShowMouseCursor выводит на экран курсор мыши. При этом курсор перемещается синхронно с перемещениями самой мыши.

Убрать (сделать невидимым) курсор мыши

Функция HideMouseCursor убирает курсор мыши с экрана. Однако при этом драйвер мыши продолжает отслеживать ее перемещения, причем к этой функции возможны вложенные вызовы. Каждый вызов функции HideMouseCursor уменьшает значение внутреннего счетчика драйвера на единицу, каждый вызов функции SliowMouseCursor увеличивает счетчик. Курсор мыши виден только, когда значение счетчика равно 0 (изначально счетчик равен -1).

При работе с мышью следует иметь в виду, что выводить изображение поверх курсора мыши нельзя. Поэтому, если нужно произвести вывод на экран в том месте, где может находиться курсор мыши, следует убрать его с экрана, выполнить требуемый вывод и затем снова вывести курсор мыши на экран.

Прочесть состояние мыши (ее координаты и состояние кнопок)

Функция ReadMouse State возвращает состояние мыши в полях структуры MouseState. Поля x и y содержат текущие координаты курсора в пикселах, поле Buttons определяет, какие кнопки нажаты. Установленный бит 0 соответствует нажатой левой кнопке, бит 1 -Правой, и бит 2 - средней.

Передвинуть курсор мыши в точку с заданными координатами

Функция MoveMouseCursor служит для установки курсора мыши в точку с заданными координатами.

Установка области перемещения курсора

При необходимости можно ограничить область перемещения мыши по экрану. Для задания области возможного перемещения курсора по горизонтали служит функция SetHorzMouseRange, для задания области перемещения по вертикали - функция SetVert Mouse Range.

Задание формы курсора

В графических режимах высокого разрешения (640 на 350 пикселов и выше) курсор задается двумя масками 16 на 16 бит и смещением координат курсора от верхнего левого угла масок. Каждую из масок можно трактовать как изображение, составленное из пикселов белого(соответствующий бит равен 1) и черного (соответствующий бит равен 0) цветов. При выводе курсора на экран сначала на содержимое экрана накладывается (с использованием операции AND_PUT) первая маска, называемая иногда AND-маской, а затем на то же самое место накладывается вторая маска (с использованием операции XOR_PUT). Все необходимые параметры для задания курсора мыши содержатся в полях структуры CursorShape. Устанавливается форма курсора при помощи функции SetMouseShape.

Установка области гашения

Иногда желательно задать на экране область, при попадании в которую курсор мыши автоматически гасится. Для этого используется функция SetHideRange. Но при выходе курсора из области гашения он не восстанавливается. Поэтому для восстановления нормальной работы курсора необходимо вызвать функцию ShowMouseCursor, независимо от того, попал ли курсор в область гашения или нет.

Установка обработчика событий

Вместо того, чтобы все время опрашивать драйвер мыши, можно передать драйверу адрес функции, которую нужно вызывать при наступлении заданных событий. Для установки этой функции следует воспользоваться функцией SetMouseHandler, где в качестве первого параметра выступает указатель на функцию, а второй параметр задает события, при наступлении которых следует вызвать переданную функцию. События задаются посредством битовой маски. Возможные события определяются при помощи символических констант MOUSE_MOVE_MASK, MOUSE_LBUTTON_PRESS и других. Требуемые условия соединяются побитовой операцией ИЛИ. Передаваемая функция получает 4 параметра - маску события, повлекшего за собой вызов функции, маску состояния кнопок мыши и текущие координаты курсора. По окончании работы программы необходимо обязательно убрать обработчик событий (при помощи функции RemoveMouseHandler).

Ниже приводится пример простейшей программы, устанавливающей обработчик событий мыши на нажатие правой кнопки мыши и задающей свою форму курсора.

```
// File Example1.cpp
#include <bios.h>
#include <conio.h>
#include "Mouse.h"

CursorShape c = {
   0x0FFF, 0x07FF, 0x01FF, 0x007F, 0x801F, 0xC007, 0xC001, 0xE000,
   0xE0FF, 0xF0FF, 0xF0FF, 0xF8FF, 0xF8FF, 0xFCFF, 0xFCFF, 0xFEFF,
   0x0000, 0x6000, 0x7800, 0x3E00, 0x3F80, 0x1FE0, 0x1FF8, 0x0FFE,
   0x0F00, 0x0700, 0x0700, 0x0300, 0x0300, 0x0100, 0x0100, 0x0000,
   1, 1
};

int      DoneFlag = 0;

void     SetVideoMode ( int mode )
{
   asm {
         mov   ax, mode
         int   10h
   }
}

#pragma argsused
void     WaitPress ( int mask, int button, int x, int y )
{
   if ( mask & MOUSE_RBUTTON_PRESS )
         DoneFlag = 1;
}

main ()
{
   SetVideoMode ( 0x12 );
   ResetMouse ();
   ShowMouseCursor ();
   SetMouseShape ( c );
   SetMouseHandler ( WaitPress );
```

```
        MoveMouseCursor ( 0, 0 );

        while ( !DoneFlag )
              ;

        HideMouseCursor ();
        RemoveMouseHandler ();
        SetVideoMode ( 3 );
    }
```

Принтер

В качестве устройства для получения "твердой" копии изображения на экране обычно выступает принтер. Практически любой принтер позволяет осуществить построение изображения, так как сам выводит символы, построенные из точек (каждый символ представляется матрицей точек; для большинства матричных принтеров - матрицей размера 8 на 11).

Для осуществления управления принтером существует специальный набор команд (обычно называемых Esc-последовательностями), позволяющий управлять режимом работы принтера, прогонкой бумаги на заданное расстояние и печати графической информации. Каждая команда представляет собой некоторый набор символов (кодов), просто посылаемых для печати на принтер. Чтобы принтер мог отличить эти команды от обычного печатаемого текста, они, как правило, начинаются с символа с кодом меньше 32, то есть кода, которому не соответствует ни одного ASCII-символа. Для большинства команд в качестве такового выступает символ Escape (код 27). Совокупность подобных команд образует язык управления принтером.

Как правило, каждый принтер имеет свои особенности, которые, естественно, находят отражение в наборе команд. Однако можно выделить некоторый набор команд, реализованный на достаточно широком классе принтеров.

9-игольчатые принтеры

Рассмотрим класс 9-игольчатых принтеров типа EPSON, STAR и совместимых с ними. Ниже приводится краткая сводка основных команд для этого класса принтеров.

Мнемоника	*Десятичный код*	*Комментарий*
LF	10	Переход на следующую строку, каретка не возвращается к началу строки
CR	13	Возврат каретки к началу строки
FF	12	Прогон бумаги до начала следующей страницы
Esc A n	27, 65, n	Установить расстояние между строками (величину прогона бумаги по команде LF) в n/72 дюйма
Esc J n	27, 74, n	Передвинуть бумагу на n/216 дюйма
Esc K n1 n2 data	27, 75, n1	Печать блока графики высотой 8 пикселов и шириной n2*256+n1 пикселов с нормальной плотностью (80 точек на дюйм)
Esc L n1 n2 data	27, 76, n1	Печать блока графики высотой 8 пикселов и шириной n2*256+n1 пикселов с двойной плотностью (120 точек на дюйм)
Esc 3 n	27, 51, n	Установка расстояния между строками для последующих команд перевода строки. Расстояние устанавливается равным n/216 дюйма

Возможные режимы вывода графики задаются в следующей таблице.

Значение m	*Режим*	*Плотность (точек на дюйм)*
0	Обычная плотность	60
1	Двойная плотность	120
2	Двойная плотность, двойная скорость	120
3	Четверная плотность	240
4	CRT I	80
5	Plotter Graphics	72
6	CRT II	90
7	Plotter Graphics, двойная плотность	144

Например, для возврата каретки в начальное положение и сдвиг бумаги на 5/216 дюйма нужно послать на принтер следующие байты: 13, 27, 74, 5.

Первый байт обеспечивает возврат каретки, а три следующих -сдвиг бумаги.

При печати графического изображения головка принтера за один проход рисует блок (изображение) шириной n1+256*n2 точек и высотой 8 точек. После n2 идут байты, задающие изображение, - по 1 байту на каждые 8 вертикально стоящих пикселов. Если точку нужно ставить в i-м снизу пикселе, то i-й бит в байте равен 1.

Пример.

128			*		*		*			
64		*						*		
32	*								*	
16		*						*		
8			*		*		*			
4					*					
2	*		*		*		*		*	
1					*					
Всего	34	80	138	0	143	0	138	80	34	0

Рассмотрим, как формируются байты для этой команды. Так как ширина изображения равна 10, то отсюда n1=10 % 256, n2=10 / 256. Для формирования первого байта, описывающего изображение, возьмем первый столбец из 8 пикселов и закодируем его битами: точке поставим в соответствие 1, а пустому месту - 0. Получившиеся биты запишем сверху вниз. При этом получается двоичное число 00100010, десятичное значение которого равно 34. Второй столбец кодируется набором бит 01010000 с десятичным значением 80. Проведя аналогичные расчеты, получим, что для печати этого изображения на принтер необходимо послать следующие коды: 27, 75, 10, 0, 34, 80, 138, 0, 143, 0, 138, 80, 34, 0.

Для вывода на принтер изображения высотой больше 8 пикселов оно предварительно разбивается на полосы высотой по 8 пикселов.

Ниже приводится пример программы, копирующей изображение экрана на 9-игольчатый матричный принтер.

```
// File Example2.cpp
#include <bios.h>
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

int      Port = 0;         // use LPT1:

inline   int   Print ( char byte )
{
```

```
    return biosprint ( 0, byte, Port );
}

void     PrintScreenFX ( int x1, int y1, int x2, int y2 )
{
    int   NumPasses = ( y2 >> 3 ) - ( y1 >> 3 ) + 1;
    int   NumCols   = x2 - x1 + 1;
    int   Byte;

    Print ( '\r' );
    for ( int pass = 0, y = y1; pass < NumPasses; pass++, y += 8 )
    {
          Print ( '\x1B ');
          Print ( 'L' );
          Print ( NumCols & 0xFF );
          Print ( NumCols >> 8 );

          for ( int x = x1; x <= x2; x++ )
          {
                Byte = 0;
                for ( int i = 0; i < 8 && y + i <= y2; i++ )
                      if ( getpixel ( x, y + i ) > 0 )
                           Byte |= 0x80 >> i;

                Print ( Byte );
          }

          Print ( '\x1B' );
          Print ( 'J' );
          Print ( 24 );
          Print ( '\r' );
    }
}

main ()
{
    int   driver = DETECT;
    int   mode;
    int   res;

    initgraph ( &driver, &mode, "" );
    if ( ( res = graphresult () ) != grOk )
    {
          printf("\nGraphics error: %s\n", grapherrormsg ( res) );
          exit ( 1 );
    }

    line ( 0, 0, 0, getmaxy () );
    line ( 0, getmaxy (), getmaxx (), getmaxy () );
    line ( getmaxx (), getmaxy (), getmaxx (), 0 );
    line ( getmaxx (), 0, 0, 0 );

    for ( int i = TRIPLEX_FONT; i <= GOTHIC_FONT; i++ )
    {
          settextstyle ( i, HORIZ_DIR, 5 );
          outtextxy ( 100, 50*i, "Some string" );
    }

    getch ();

    PrintScreenFX ( 0, 0, getmaxx (), getmaxy () );

    closegraph ();
}
```

24-игольчатые (LQ) принтеры

Язык управления для большинства 24-игольчатых принтеров является надмножеством над языком для 9-игольчатых принтеров, поэтому все приведенные ранее команды будут работать и с LQ-принтерами (используя только 8 игл, а не 24). Для использования всех 24 игл предусмотрены дополнительные режимы в команде Esc '*'.

Значение m	*Режим*	*Плотность (точек на дюйм)*
32	Обычная плотность	60
33	Двойная плотность	120
38	CRT III	90
39	Тройная плотность	180

При этом количество столбцов пикселов, как и раньше, равно n1 + 256*n2, но для каждого столбца задается уже 3 байта.

Большинство струйных принтеров на уровне языка управления совместимы с LQ-принтерами.

Лазерные принтеры

Одним из наиболее распространенных классов лазерных принтеров являются лазерные принтеры серии HP LaserJet фирмы Hewlett Packard. Все они управляются языком PCL. Отметим, что большое количество лазерных принтеров других фирм также поддерживают язык PCL. Ниже приводится краткая сводка основных команд этого языка, используемых при выводе графики.

Мнемоника	*Десятичный код*	*Комментарии*
Esc * t 75 R	27, 42, 116, 55, 53, 82	Установка плотности печати 75 точек на дюйм
Esc * t 100 R	27,42, 116,49, 48,48, 82	Установка плотности печати 100 точек на дюйм
Esc * t 150 R	27, 42, 116, 49, 53,48, 82	Установка плотности печати 150 точек на дюйм
Esc * t 300 R	27,42, 116, 51,48,48,82	Установка плотности печати 300 точек на дюйм
Esc & a # R	27, 38, 97, #...#, 82	Вертикальное позиционирование
Esc & a # C	27, 38, 97, #...#, 67	Горизонтальное позиционирование
Esc * r 1 A	27,42, 114,49, 65	Начать вывод графики
Esc * b # W data	27, 42, 98, #...#, 87, data	Передать графические данные
Esc * r B	27, 42, 114, 66	Закончить вывод графики

Здесь символ # означает, что в этом месте выводятся цифры, задающие десятичное значение числа. Пикселы собираются в байты по горизонтали, то есть за одну команду Esc * b передается сразу целая строка пикселов.

Ниже представлена программа, копирующая содержимое экрана на лазерный принтер, поддерживающий язык PCL.

```
// File Example3.cpp
#include <bios.h>
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>

int      Port = 0;          // use LPT1:

inline   int   Print ( char byte )
{
   return biosprint ( 0, byte, Port );
}

int      PrintStr ( char * str )
```

```
{
   int   st;

   while ( *str != '\0' )
         if ( ( st = Print ( *str++ ) ) & 1 )
               return st;

   return 0;
}

void     PrintScreenLJ ( int x1, int y1, int x2, int y2 )
{
   int   NumCols   = x2 - x1 + 1;
   int   Byte;
   char  str [20];

   PrintStr ( "\x1B*t150R" );           // set density 150 dpi
   PrintStr ( "\x1B&a5C" );       // move cursor to col 5
   PrintStr ( "\x1B*r1A" );       // begin raster graphics

   sprintf ( str, "\x1B*b%dW", (NumCols+7)>>3);     // prepare line header

   for ( int y = y1; y <= y2; y++ )
   {
         PrintStr ( str );

         for ( int x = x1; x <= x2; )
         {
               Byte = 0;
               for ( int i = 0; i < 8 && x <= x2; i++, x++ )
                     if ( getpixel ( x, y ) > 0 )
                           Byte |= 0x80 >> i;

               Print ( Byte );
         }
   }

   PrintStr ( "\x1B*rB" );
}

main ()
{
   int   driver = DETECT;
   int   mode;
   int   res;

   initgraph ( &driver, &mode, "" );
   if ( ( res = graphresult () ) != grOk )
   {
         printf("\nGraphics error: %s\n", grapherrormsg ( res) );
         exit ( 1 );
   }

   line ( 0, 0, 0, getmaxy () );
   line ( 0, getmaxy (), getmaxx (), getmaxy () );
   line ( getmaxx (), getmaxy (), getmaxx (), 0 );
   line ( getmaxx (), 0, 0, 0 );

   for ( int i = TRIPLEX_FONT; i <= GOTHIC_FONT; i++ )
   {
         settextstyle ( i, HORIZ_DIR, 5 );
         outtextxy ( 100, 50*i, "Some string" );
   }

   getch ();
```

```
    PrintScreenLJ ( 0, 0, getmaxx (), getmaxy () );

    closegraph ();
}
```

Видеокарты EGA и VGA

Основным графическим устройством, с которым чаще всего приходится работать, является видеосистема компьютера. Обычно она состоит из видеокарты (адаптера) и подключенного к ней монитора. Изображение хранится в растровом виде в памяти видеокарты: аппаратура карты обеспечивает регулярное (50-70 раз в сек.) чтение этой Памяти и отображение ее на экране монитора. Поэтому вся работа с изображением сводится к тем или иным операциям с видеопамятью. Наиболее распространенными видеокартами сейчас являются Клоны карт EGA (Enhanced Graphics Adaptor) и VGA (Video Graphics Array). Кроме того, существует большое количество различных SVGA-карт, которые будут рассмотрены в конце главы.

Приведем список основных режимов для этих карт. Режим определяется номером, разрешением экрана и количеством цветов.

Номер режима	*Разрешение экрана*	*Количество цветов*
0Dh	320x200	16
0Eh	640x200	16
0Fh	640x350	2
10h	640x350	16
1 1 h (VGA)	640x480	2
12h (VGA)	640x480	16
13h (VGA)	320x200	256

Каждая видеоплата содержит в своем составе собственный BIOS для работы с ней и поддержки основных функций платы.

Ниже приводится файл, содержащий базовые функции по работе с графикой, доступные через BIOS.

```
// File Ega.Cpp
#include  <dos.h>
#include  "Ega.h"

intFindEGA ()
{
    asm {
          mov    ax, 1200h
          mov    bx, 10h
          int    10h
    }

    return _BL != 0x10;
}

intFindVGA ()
{
    asm {
          mov    ax, 1A00h
          int    10h
    }

    return _AL == 0x1A;
}

void      SetVideoMode ( int mode )
{
```

```
    asm {
          mov    ax, mode
          int    10h
    }
}

void      SetVisiblePage ( int page )
{
    asm {
          mov    ah, 5
          mov    al, byte ptr page
          int    10h
    }
}

char far * FindROMFont ( int size )
{
    int    b = ( size == 16 ? 6 : 2 );

    asm {
          push   es
          push   bp
          mov    ax, 1130h
          mov    bh, byte ptr b
          mov    bl, 0
          int    10h
          mov    ax, es
          mov    bx, bp
          pop    bp
          pop    es
    }

    return (char far *) MK_FP ( _AX, _BX );
}

void      SetPalette ( RGB far * Palette, int size )
{
    asm {
          push   es
          mov    ax, 1012h
          mov    bx, 0             // first color to set
          mov    cx, size          // # of colors
          les    dx, Palette       // ES:DX == table of color values
          int    10h
          pop    es
    }
}
```

Функции FindEGA и FindVGA позволяют определить наличие EGA- или VGA-совместимой видеокарты.

Для установки нужного режима можно воспользоваться процедурой SetVideoMode.

Функция FindROMFont возвращает адрес системного шрифта заданного размера (8, 14 или 16 пикселов высоты).

Функция Set Palette служит для установки палитры и является аналогом функции setrgbpalette.

16-цветные режимы адаптеров EGA и VGA

Для 16-цветных режимов под каждый пиксел изображения необходимо выделить 4 бита видеопамяти (2 = 16). Однако эти 4 бита выделяются не последовательно в одном байте, а разнесены в 4 разных блока (цветовые плоскости) видеопамяти.

Вся видеопамять карты (обычно 256 Кбайт) делится на 4 равные части, называемые цветовыми плоскостями. Каждому пикселу ставится в соответствие по одному биту в каждой плоскости, причем все эти биты одинаково расположены относительно ее начала. Обычно эти плоскости представляют параллельно расположенными одна над другой, так что каждому пикселу соответствует 4 расположенных друг под другом бита. Все эти плоскости проектируются на один и тот же участок адресного пространства процессора, начиная с адреса 0xA000:0. При этом все операции чтения и записи видеопамяти опосредуются видеокартой! Поэтому, если вы записали байт по адресу 0xA000:0, то это вовсе не означает, что посланный байт в действительности запишется хотя бы в одну из этих плоскостей, точно так же как при операции чтения прочитанный байт не обязательно будет совпадать с одним из 4 байтов в соответствующих плоскостях. Механизм этого опосредования определяется логикой карты, но для программиста существует возможность известного управления этой логикой (при работе одновременно с 8 пикселами).

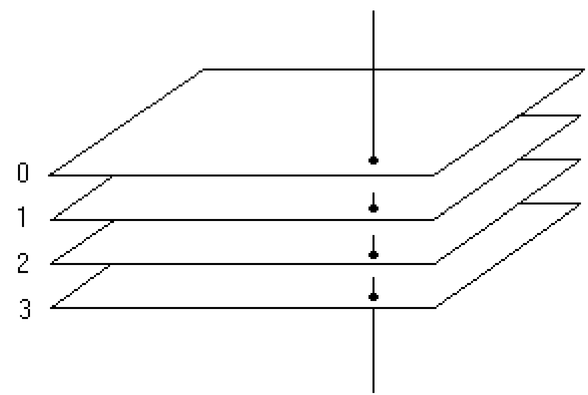


Рис. 1

Для работы с пикселом необходимо определить адрес байта в видеопамяти, содержащего данный пиксел, и позицию пиксела внутри байта (поскольку один пиксел отображается на один бит в каждой плоскости, то байт соответствует сразу 8 пикселам).

Поскольку' видеопамять под пикселы отводится последовательно слева направо и сверху вниз, то одна строка соответствует 80 байтам адреса и каждым 8 последовательным пикселам, начинающимся с позиции, кратной 8, соответствует один байт. Тем самым адрес байта задается выражением 80*y+(x>>3), а его номер внутри байта задается выражением x&7, где (x, y) - координаты пиксела.

Для идентификации позиции пиксела внутри байта часто используется не номер бита, а битовая маска - байт, в котором отличен от нуля только бит, стоящий на позиции пиксела.

Битовая маска задается следующим выражением: 0x80>>(x&7).

На видеокарте находится набор специальных 8-битовых регистров. Часть из них доступна только для чтения, часть - только для записи, а некоторые вообще недоступны программисту. Доступ к регистрам осуществляется через порты ввода/вывода процессора.

Регистры видеокарты делятся на несколько групп. При этом каждой группе соответствует пара последовательных портов (порт адреса порт значения). Для записи значения в регистр видеокарты необходимо сначала записать номер регистра в первый порт (порт адреса), а затем записать значение в следующий порт (порт значения). Для гения регистра в порт адреса записывается номер регистра, а затем его значение читается из порта значения.

Ниже приводится файл, определяющий необходимые константы inline-функции для работы с портами видеокарты. Функции WriteReg и ReadReg служат для доступа к регистрам.

```
// File Ega.h
#ifndef  __EGA__
#define  __EGA__

#include <dos.h>

#define  EGA_GRAPHICS0x3CE        // Graphics Controller base addr
#define  EGA_SEQUENCER     0x3C4        // Sequencer base addr
#define  EGA_CRTC    0x3D4
```

```
#define  EGA_SET_RESET           0
#define  EGA_ENABLE_SET_RESET    1
#define  EGA_COLOR_COMPARE  2
#define  EGA_DATA_ROTATE         3
#define  EGA_READ_MAP_SELECT     4
#define  EGA_MODE          5
#define  EGA_MISC          6
#define  EGA_COLOR_DONT_CARE     7
#define  EGA_BIT_MASK      8

#define  EGA_MAP_MASK      2

struct   RGB
{
   char  Red;
   char  Green;
   char  Blue;
};

inline   void  WriteReg ( int base, int reg, int value )
{
   outportb ( base,     reg );
   outportb ( base + 1, value );
}

inline   char  ReadReg ( int base, int reg )
{
   outportb ( base, reg );

   return inportb ( base + 1 );
}

inline   char  PixelMask ( int x )
{
   return 0x80 >> ( x & 7 );
}

inline   char  LeftMask ( int x )
{
   return 0xFF >> ( x & 7 );
}

inline   char  RightMask ( int x )
{
   return 0xFF << ( 7 ^ ( x & 7 ) );
}

inline   void  SetRWMode ( int ReadMode, int WriteMode )
{
   WriteReg ( EGA_GRAPHICS, EGA_MODE, ( WriteMode & 3 ) | ( ( ReadMode & 1 ) << 3 ) );
}

inline   void  SetWriteMode ( int mode )
{
   WriteReg ( EGA_GRAPHICS, EGA_DATA_ROTATE, ( mode & 3 ) << 3 );
}

int         FindEGA ();
int         FindVGA ();
void        SetVideoMode ( int );
void        SetVisiblePage ( int );
char far * FindROMFont ( int );
void        SetPalette ( RGB far * Palette, int );

#endif
```

Рассмотрим две основные группы регистров, принадлежащих двум частям видеокарты, - Graphics Controller и Sequencer. Каждой группе соответствует своя пара портов.

Graphics Controller (порты 3CE- 3CF)

Номер	*Регистр*	*Стандартное значение*
0	Set/Reset	00
1	Enable Set/Reset	00
2	Color Compare	00
3	Data rotate	00
4	Read Map Select	00
5	Mode	10
6	Miscellaneous	05
7	Color Don't Care	0F
8	Bit Mask	FF

Для записи в регистр необходимо сначала послать номер регистра в порт 3CE, а затем записать соответствующее значение в порт 3CF.

Для EGA-карты все эти регистры доступны только для чтения, VGA-адаптер поддерживает и запись, и чтение.

Проиллюстрируем это на процессе установки регистра битовой маски (Bit Mask) (установка остальных регистров аналогична).

```
void SetBitMask ( char mask ) WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, mask );
```

Sequencer (порты 3C4-3C5)

Из всех регистров этой группы мы рассмотрим только регистр маски плоскости (Map Mask) и номер 2.

Процедура SetMapMask устанавливает значение регистра маски плоскости.

Рассмотрим теперь, как происходит работа с видеопамятью.

При операции чтения байта из видеопамяти читаются сразу 4 байта - по одному из каждой плоскости. При этом прочитанные значения записываются в специальные регистры - "защелки" (latch-регистры), недоступные для прямого доступа. Байт, прочитанный процессором, является комбинацией значений latch-регистров.

При операции записи посланный процессором байт накладывается на значения latch-регистров по правилам, определяемым значениями других регистров, а результирующие 4 байта записываются в соответствующие плоскости.

Так как при записи используются значения latch-регистров, то часто необходимо, чтобы перед записью в них находились исходные значения тех байтов, которые затем изменяются. Это часто приводит к необходимости осуществлять чтение байта по адресу перед записью по этому адресу нового значения.

Правила, определяющие наложение при записи посланных процессором данных на значения latch-регистров, определяются установленным режимом записи, и, соответственно, режим чтения задает способ, которым определяется значение, прочитанное процессором.

Видеокарта EGA поддерживает два режима чтения и три режима записи, у карты VGA есть еще один дополнительный режим записи.

Установка режимов чтения и записи осуществляется записью соответствующих значений в регистр Mode. Бит 3 отвечает за режим чтения, биты 0 и 1 - за режим записи.

Функция SetRWMode служит для установки режимов чтения и записи.

Режимы чтения

Режим чтения 0

В этом режиме возвращается байт из latch-регистра (плоскости) с номером из регистра Read Map Select.

В приведенном примере возвращается значение (цвет) пиксела с координатами (x, y). Для этого с каждой из плоскостей по очереди читаются биты и из них собирается цветовое значение пиксела.

```
// File ReadPxl.cpp
int     ReadPixel ( int x, int y )
{
    int      color = 0;
    char far * vptr = (char far *) MK_FP ( 0xA000, y*80 + (x >> 3) );
    char     mask = PixelMask ( x );

    for ( int plane = 3; plane >= 0; plane-- )
    {
          WriteReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, plane );
          color <<= 1;
          if ( *vptr & mask )
                color |= 1;
    }

    return color;
}
```

Режим чтения 1

В возвращаемом значении i-й бит равен единице, если

```
GetPixel & ColorDon'tCare = ColorCompare & ColorDon'tCare
```

В случае, если ColorDon'tCare == 0F, в прочитанном байте в тех позициях, где цвет пиксела совпадает со значением в регистре ColorCompare, будет стоять единица.

Этот режим очень удобен для поиска точек заданного цвета.

Приведенная процедура осуществляет поиск пиксела цвета Color В строке у начиная с позиции х. При этом используется режим чтения 1. Все байты, соответствующие данной строке, читаются по очереди, и, как только будет получено ненулевое значение (найден по крайней мере I пиксел данного цвета в байте), оно возвращается.

```
// File FindPxl.cpp
int     FindPixel ( int x1, int x2, int y, int color )
{
    char far * vptr  = (char far *) MK_FP ( 0xA000, y*80 + (x1 >> 3) );
    int      cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
    char     lmask = LeftMask ( x1 );
    char     rmask = RightMask ( x2 );
    char     mask;

    SetRWMode ( 1, 0 );
    WriteReg ( EGA_GRAPHICS, EGA_COLOR_COMPARE, color );

    if ( cols < 0 )
          return *vptr & lmask & rmask;

    if ( mask = *vptr++ & lmask )
          return mask;

    while ( cols-- > 0 )
          if ( mask = *vptr++ )
```

```
            return mask;

    return *vptr & rmask;
}
```

Режимы записи

Режим записи 0

Это, пожалуй, самый сложный из всех рассматриваемых режимов, дающий, однако, самые большие возможности.

В рассматриваемом режиме регистр BitMask позволяет защищать от изменения определенные пикселы. В тех позициях, где соответствующий бит из регистра BitMask равен нулю, пиксел не изменяет своего значения. Регистр MapMask позволяет защищать от изменения определенные плоскости.

Биты 3 и 4 регистра DataRotate определяют способ наложения выводимого изображения на существующее (аналогично функции setwritemode).

Значение битов	*Операция*	*Эквивалент в BGI*
0 0	Замена	COPY PUT
0 1	Or	OR PUT
1 0	And	AND PUT
1 1	Xor	XOR_PUT

Процедура SetWriteMode устанавливает соответствующий режим наложения.

Посланный процессором байт циклически сдвигается вправо на указанное в битах 0-2 регистра Data Rotate количество раз.

Результирующее значение определяется следующим образом. На плоскость, соответствующий бит которой в регистре Enable Set/Reset равен нулю, накладывается посланный процессором байт, "прокрученный" заданное количество раз с учетом регистров BitMask и MapMask. Если соответствующий бит равен единице, то во все Позиции, разрешенные регистром BitMask, записывается бит из регистра Set/Reset, соответствующий плоскости.

На практике наиболее часто встречаются следующие два случая:

1. Enable Set/Reset = 0 (байт, посланный процессором, циклически сдвигается в соответствии со значением битов 0-2 регистра Data Rotate; после этого получившийся байт накладывается заданным способом (см. биты 3-4 регистра Data Rotate) на те плоскости, которые разрешены регистром Map Mask, причем изменяются лишь разрешенные регистром BitMask биты).

2. Enable Set/Reset = OF (в позиции, разрешенные регистром BitMask, ставятся точки цвета, заданного в регистре Set/Reset; байт, посланный процессором, никакой роли не играет).

Для того, чтобы нарисовать только нужный пиксел, необходимо поставить регистр BitMask так, чтобы защитить от изменения остальные 7 пикселов, соответствующих этому байту.

```
// File WritePxl.cpp
void     WritePixel ( int x, int y, int color )
{
   char far * vptr = (char far *) MK_FP ( 0xA000, y*80 + (x >> 3) );

   WriteReg ( EGA_GRAPHICS, EGA_ENABLE_SET_RESET, 0x0F ); // enable all planes
   WriteReg ( EGA_GRAPHICS, EGA_SET_RESET, color );
```

```
    WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, PixelMask ( x ) );

    *vptr += 1;

    WriteReg ( EGA_GRAPHICS, EGA_ENABLE_SET_RESET, 0 );    // disable all planes
    WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );         // restore reg
}
```

Режим записи 1

В этом режиме значения latch-регистров непосредственно копируются в соответствующие плоскости. Регистры масок и режима не действуют. Посланное процессором значение не играет никакой роли. Этот режим позволяет осуществлять быстрое копирование фрагментов видеопамяти. При чтении байта по исходному адресу прочитанные 4 байта с плоскостей загружаются в latch-регистры, а при записи значения latch-регистров записываются в плоскости по адресу, по которому шла запись. Таким образом, за одну операцию перезаписи копируется сразу 4 байта (8 пикселов).

Приведенная ниже функция осуществляет копирование прямоугольной области экрана в соответствующую область с верхним левым утлом в точке (x, y).

В силу ограничений режима записи 1 эта процедура может копировать только области, где x1 кратно 8 и ширина кратна 8, так как копирование осуществляется блоками по 8 пикселов сразу. Кроме того, этот пример не учитывает возможности того, что область, куда производится копирование, имеет непустое пересечение с исходной областью. В этом случае возможна некорректная работа процедуры, и, чтобы подобного не возникало, необходимо проверять области на пересечение: при непустом пересечении осуществляется копирование в обратном порядке.

```
 // File copyrect.cpp
void     CopyRect ( int x1, int y1, int x2, int y2, int x, int y )
{
    char far * src  = (char far *) MK_FP ( 0xA000, y1*80 + (x1 >> 3) );
    char far * dst  = (char far *) MK_FP ( 0xA000, y*80 + (x >> 3) );
    int      cols  = ( x2 >> 3 ) - ( x1 >> 3 );

    SetRWMode ( 0, 1 );

    for ( int i = y1; i <= y2; i++ )
    {
          for ( int j = 0; j < cols; j++ )
                *dst++ = *src++;

          src += 80 - cols;
          dst += 80 - cols;
    }

    SetRWMode ( 0, 0 );
}
```

Режим записи 2

В этом режиме младшие 4 бита байта, посланного процессором, определяют цвет, которым будут построены не защищенные битовой маской пикселы. Регистр битовой маски защищает от изменения определенные пикселы. Регистр маски плоскости защищает от изменения определенные плоскости. Регистр DataRotate устанавливает способ наложения построенных пикселов на существующее изображение.

Приведенный ниже пример рисует прямоугольник заданного цвета, используя режим записи 2.

```
// File bar.cpp
void     Bar ( int x1, int y1, int x2, int y2, int color )
{
```

```
    char far * vptr  = (char far *) MK_FP ( 0xA000, y1*80 + (x1 >> 3) );
    int      cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
    char     lmask = LeftMask ( x1 );
    char     rmask = RightMask ( x2 );
    char     latch;

    SetRWMode ( 0, 2 );

    if ( cols < 0 )          // both x1 & x2 are located in the same byte
    {
          WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, lmask & rmask );

          for ( int y = y1; y <= y2; y++, vptr += 80 )
          {
                latch = *vptr;
                *vptr = color;
          }

          WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
    }
    else
    {
          for ( int y = y1; y <= y2; y++ )
          {
                WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, lmask );
                latch = *vptr;
                *vptr++ = color;

                WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
                for ( int x = 0; x < cols; x++ )
                {
                      latch = *vptr;
                      *vptr++ = color;
                }

                WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, rmask );
                latch = *vptr;
                *vptr++ = color;

                vptr += 78 - cols;
          }
    }

    SetRWMode ( 0, 0 );
    WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );
}
```

Следующие две функции служат для запоминания и восстановления записанного изображения.

```
// File store.cpp
void     StoreRect ( int x1, int y1, int x2, int y2, char huge * buf )
{
    char far * vptr  = (char far *) MK_FP ( 0xA000, y1*80 + (x1 >> 3) );
    int      cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;

    if ( cols < 0 )
          cols = 0;

    for ( int y = y1; y <= y2; y++, vptr += 80 )
          for ( int plane = 0; plane < 4; plane++ )
          {
                WriteReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, plane );

                for ( int x = 0; x < cols + 2; x++ )
                      *buf++ = *vptr++;
```

```
                vptr -= cols + 2;
        }
}

void     RestoreRect ( int x1, int y1, int x2, int y2, char huge * buf )
{
   char far * vptr  = (char far *) MK_FP ( 0xA000, y1*80 + (x1 >> 3) );
   int      cols  = ( x2 >> 3 ) - ( x1 >> 3 ) - 1;
   char     lmask = LeftMask ( x1 );
   char     rmask = RightMask ( x2 );
   char     latch;

   if ( cols < 0 )
   {
        lmask &= rmask;
        rmask  = 0;
        cols   = 0;
   }

   for ( int y = y1; y <= y2; y++, vptr += 80 )
        for ( int plane = 0; plane < 4; plane++ )
        {
              WriteReg ( EGA_GRAPHICS,  EGA_BIT_MASK, lmask );
              WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << plane );

              latch = *vptr;
              *vptr++ = *buf++;

              WriteReg ( EGA_GRAPHICS, EGA_BIT_MASK, 0xFF );

              for ( int x = 0; x < cols; x++ )
                    *vptr++ = *buf++;

              WriteReg ( EGA_GRAPHICS,  EGA_BIT_MASK, rmask );

              latch = *vptr;
              *vptr++ = *buf++;

              vptr -= cols + 2;
        }

        WriteReg ( EGA_GRAPHICS,  EGA_BIT_MASK, 0xFF );
        WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
```

256-цветный режим адаптера VGA

Из всех видеорежимов этот режим является самым простым. При разрешении экрана 320*200 точек он позволяет одновременно использовать все 256 цветов. Для одновременного отображения 256 цветов Необходимо под каждую точку на экране отвести по 8 бит. В рассматриваемом режиме эти 8 бит идут последовательно один за другим, Образуя 1 байт. Тем самым в этом режиме плоскости не используются. Видеопамять начинается с адреса 0xA000:0. При этом точке с координатами (x, y) соответствует байт памяти по адресу 320y + x.

```
void WritePixel ( int x, int y, int color )
{
   pokeb(0xA000, 320*y + x, color);
}
int ReadPixel(int x, int y)
{
   return peekb ( 0xA000, 320*y + x );
}
```

Нестандартные режимы адаптера VGA

Для 256-цветных режимов существует еще один способ организации видеопамяти. При этом 8 бит, отводимых под каждый пиксел, также хранятся вместе, образуя 1 байт, но эти байты находятся на разных плоскостях видеопамяти.

Пиксел	*Адрес*	*Плоскость*
(0,0)	0	0
(1,0)	0	1
(2,0)	0	2
(3,0)	0	3
(4,0)	1	0
(5,0)	1	1
...	...	...
(x, y)	y * 80 + (x >> 2)	x & 3

В этом режиме сохраняются все свойства основных регистров и механизм их действия за исключением того, что меняется интерпретация находящихся в видеопамяти значений. Режим позволяет за одну операцию изменить сразу до четырех пикселов. Еще одним преимуществом этого режима является возможность работы с несколькими страницами видеопамяти, недоступная в стандартном 256-цветном режиме.

Ниже приводится программа, устанавливающая режим с разрешением 320 на 200 пикселов с использованием 256 цветов посредством изменения стандартного режима 13h, и иллюстрируется возможность работы сразу с четырьмя страницами.

```
#include    <alloc.h>
#include <conio.h>
#include <mem.h>
#include <stdio.h>
#include "Ega.h"

unsigned   PageBase = 0;
char       LeftPlaneMask  [] = { 0x0F, 0x0E, 0x0C, 0x08 };
char        RightPlaneMask [] = { 0x01, 0x03, 0x07, 0x0F };
char far * Font;

void     SetX ()
{
   SetVideoMode ( 0x13 );

   PageBase = 0xA000;

   WriteReg ( EGA_SEQUENCER, 4, 6 );
   WriteReg ( EGA_CRTC, 0x17, 0xE3 );
   WriteReg ( EGA_CRTC, 0x14, 0 );

                      // clear screen
   WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
   _fmemset ( MK_FP ( PageBase, 0 ), '\0', 0xFFFF );
}

void     SetVisualPage ( int page )
{
   unsigned addr = page * 0x4000;

                      // wait for vertical retrace
   while ( ( inportb ( 0x3DA ) & 0x08 ) == 0 )
          ;

   WriteReg ( EGA_CRTC, 0x0C, addr >> 8 );
   WriteReg ( EGA_CRTC, 0xDC, addr & 0x0F );
```

```
}

void     SetActivePage ( int page )
{
   PageBase = 0xA000 + page * 0x400;
}

void     WritePixel ( int x, int y, int color )
{
   WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << ( x & 3 ) );

   pokeb ( PageBase, y*80 + ( x >> 2 ), color );

   WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}

int      ReadPixel ( int x, int y )
{
   WriteReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, x & 3 );

   return peekb ( PageBase, y*80 + ( x >> 2 ) );
}

void     Bar ( int x1, int y1, int x2, int y2, int color )
{
   char far * vptr  = (char far *) MK_FP ( PageBase, y1*80 + (x1 >> 2) );
   char far * ptr   = vptr;
   int        cols  = ( x2 >> 2 ) - ( x1 >> 2 ) - 1;
   char       lmask = LeftPlaneMask [ x1 & 3 ];
   char       rmask = RightPlaneMask [ x2 & 3 ];

   if ( cols < 0 )          // both x1 & x2 are located in the same byte
   {
         WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, lmask & rmask );

         for ( int y = y1; y <= y2; y++, vptr += 80 )
               *vptr = color;

         WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
   }
   else
   {
         WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, lmask );

         for ( int y = y1; y <= y2; y++, vptr += 80 )
               *vptr = color;

         WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );

         vptr = ++ptr;

         for ( y = y1; y <= y2; y++, vptr += 80 - cols )
               for ( int x = 0; x < cols; x++ )
                     *vptr++ = color;

         WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, rmask );

         vptr = ptr + cols;

         for ( y = y1; y <= y2; y++, vptr += 80 )
               *vptr = color;
   }

   WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
}
```

```
void    DrawString ( int x, int y, char * str, int color )
{
   for ( ; *str != '\0'; str++, x+= 8 )
         for ( int j = 0; j < 16; j++ )
         {
               char  byte = Font [16 * (*str) + j];

               for ( int i = 0; i < 8; i++, byte <<= 1 )
                     if ( byte & 0x80 )
                           WritePixel ( x+i, y+j, color );
         }
}

main ()
{
   if ( !FindVGA () )
   {
         printf ( "\nVGA compatible card not found." );
         return -1;
   }

   SetX ();                 // set 320x200 256 colors X-mode

   Font = FindROMFont ( 16 );

   for ( int i = 0; i < 256; i++ )
         WritePixel ( i, 0, i );

   for ( i = 5; i < 140; i++ )
         Bar ( 2*i, i, 2*i+30, i+30, i );

   DrawString ( 110, 100, "Page 0", 70 );
   getch ();

   SetActivePage ( 1 );
   SetVisualPage ( 1 );
   Bar ( 10, 20, 300, 200, 33 );
   DrawString ( 110, 100, "Page 1", 75 );
   getch ();

   SetActivePage ( 2 );
   SetVisualPage ( 2 );
   Bar ( 10, 20, 300, 200, 39 );
   DrawString ( 110, 100, "Page 2", 80 );
   getch ();

   SetActivePage ( 3 );
   SetVisualPage ( 3 );
   Bar ( 10, 20, 300, 200, 44 );
   DrawString ( 110, 100, "Page 3", 85 );
   getch ();

   SetVisualPage ( 0 );
   getch ();
   SetVisualPage ( 1 );
   getch ();
   SetVisualPage ( 2 );
   getch ();

   SetVideoMode ( 3 );
}
```

Опишем процедуры, устанавливающие этот режим с нестандартными разрешениями 320 на 240 пикселов и 360 на 480 пикселов

```
void   SetX320x240 ()
{
   static int CRTCTable [] = {
         0x0D06,           // vertical total
         0x3E07,             // overflow (bit 8 of vertical counts)
         0x4109,           // cell height (2 to double-scan)
         0xEA10,           // vert sync start
         0xAC11,                 // vert sync end and protect cr0-cr7
         0xDF12,           // vertical displayed
         0x0014,           // turn off dword mode
         0xE715,           // vert blank start
         0x0616,           // vert blank end
         0xE317            // turn on byte mode
   };

   SetVideoMode ( 0x13 );

   PageBase     = 0xA000;
   BytesPerLine = 80;
   WriteReg ( EGA_SEQUENCER, 4, 6 );
   WriteReg ( EGA_CRTC, 0x17, 0xE3 );
   WriteReg ( EGA_CRTC, 0x14, 0 );

   WriteReg ( EGA_SEQUENCER, 0, 1 );    // synchronous reset
   outportb ( 0x3C2, 0xE3 );            // select 25 MHz dot clock
                              // & 60 Hz scan rate
   WriteReg ( EGA_SEQUENCER, 0, 3 );    // restart sequencer
   WriteReg ( EGA_CRTC, 0x11, ReadReg ( EGA_CRTC, 0x11 ) & 0x7F );

   for ( int i = 0; i < sizeof ( CRTCTable ) / sizeof ( int ); i++ )
         outport ( EGA_CRTC, CRTCTable [i] );

                              // clear screen
   WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
   _fmemset ( MK_FP ( PageBase, 0 ), '\0', 0xFFFF );
}

void     SetX360x480 ()
{
   static int CRTCTable [] = {
         0x6b00,
         0x5901,
         0x5A02,
         0x8E03,
         0x5E04,
         0x8A05,
         0x0D06,      // vertical total
         0x3E07,      // overflow (bit 8 of vertical counts)
         0x4009,      // cell height (2 to double-scan)
         0xEA10,      // vert sync start
         0xAC11,      // vert sync end and protect cr0-cr7
         0xDF12,      // vertical displayed
         0x2D13,
         0x0014,      // turn off dword mode
         0xE715,      // vert blank start
         0x0616,      // vert blank end
         0xE317       // turn on byte mode
   };

   SetVideoMode ( 0x13 );

   PageBase     = 0xA000;
   BytesPerLine = 90;

   WriteReg ( EGA_SEQUENCER, 4, 6 );
```

```
    WriteReg ( EGA_CRTC, 0x17, 0xE3 );
    WriteReg ( EGA_CRTC, 0x14, 0 );

    WriteReg ( EGA_SEQUENCER, 0, 1 );   // synchronous reset
    outportb ( 0x3C2, 0xE7 );           // select 25 MHz dot clock
                                  // & 60 Hz scan rate
    WriteReg ( EGA_SEQUENCER, 0, 3 );   // restart sequencer
    WriteReg ( EGA_CRTC, 0x11, ReadReg ( EGA_CRTC, 0x11 ) & 0x7F );

    for ( int i = 0; i < sizeof ( CRTCTable ) / sizeof ( int ); i++ )
          outport ( EGA_CRTC, CRTCTable [i] );

                                  // clear screen
    WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
    _fmemset ( MK_FP ( PageBase, 0 ), '\0', 0xFFFF );
  }

  void     SetVisualPage ( int page )
  {
    unsigned addr = page * 0x4B00;

                       // wait for vertical retrace
    while ( ( inportb ( 0x3DA ) & 0x08 ) == 0 )
          ;

    WriteReg ( EGA_CRTC, 0x0C, addr >> 8 );
    WriteReg ( EGA_CRTC, 0xDC, addr & 0x0F );
  }

  void     SetActivePage ( int page )
  {
    PageBase = 0xA000 + page * 0x4B0;
  }

  void     WritePixel ( int x, int y, int color )
  {
    WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 1 << ( x & 3 ) );

    pokeb ( PageBase, y * BytesPerLine + ( x >> 2 ), color );

    WriteReg ( EGA_SEQUENCER, EGA_MAP_MASK, 0x0F );
  }

  int      ReadPixel ( int x, int y )
  {
    WriteReg ( EGA_GRAPHICS, EGA_READ_MAP_SELECT, x & 3 );

    return peekb ( PageBase, y * BytesPerLine + ( x >> 2 ) );
  }
```

Программирование SVGA-адаптеров

Существует большое количество видеокарт, хотя и совместимых с VGA, но предоставляющих достаточно большой набор дополнительных режимов. Обычно такие карты называют SuperVGA или SVGA. Существует большое количество SVGA-карт различных производителей, сильно различающихся по основным возможностям и, как правило, несовместимых друг с другом. Сам термин "SVGA" обозначает скорее не стандарт (как VGA), а некоторое его расширение.

Рассмотрим работу с 256-цветными режимами SVGA-адаптеров. Почти все они построены одинаково - под каждый пиксел отводится один байт, и вся видеопамять разбивается на банки одинакового размера (обычно по 64 Кбайт), при этом область адресного

пространства 0xA000:0-0xA000:0xFFFF соответствует выбранному банку. Ряд карт позволяет работать сразу с двумя банками.

При такой организации памяти процедура WritePixel для карт с 64-килобайтовыми банками выглядит следующим образом.

```
void WritePixel ( int x, int y, int color )
```

где функция Set Bank служит для установки банка с заданным номером.

Практически все различие между картами сводится к установке I режима с заданным разрешением и установке банка с заданным номером.

Ниже приводится пример программы, работающей с режимом 640 на 480 точек при 256 цветах для SVGA Trident. При этом функция FindTrident служит для проверки того, что данный видеоадаптер действительно установлен.

```
// File Trident.Cpp
#include  <conio.h>
#include  <dos.h>

#define LOWORD(l)          ((int)(l))
#define HIWORD(l)          ((int)((l) >> 16))

static    int    CurBank = 0;

void      SetTridentMode ( int mode )
{
   asm {
          mov    ax, mode
          int    10h

          mov    dx, 3CEh       // set pagesize to 64k
          mov    al, 6
          out    dx, al
          inc    dx
          in     al, dx
          dec    dx
          or     al, 4
          mov    ah, al
          mov    al, 6
          out    dx, ax
          mov    dx, 3C4h       // set to BPS mode
          mov    al, 0Bh
          out    dx, al
          inc    dx
          in     al, dx
   }
}

void      SetTridentBank ( int start )
{
   if ( start == CurBank )
          return;

   CurBank = start;
   asm {
          mov    dx, 3C4h
          mov    al, 0Bh
          out    dx, al
          inc    dx
          mov    al, 0
          out    dx, al
          in     al,dx
          dec    dx
          mov    al, 0Eh
          mov    ah, byte ptr start
          xor    ah, 2
```

```
            out     dx, ax
    }
}

void        WritePixel ( int x, int y, int color )
{
    long    addr = 640l * (long)y + (long)x;

    SetTridentBank ( HIWORD ( addr ) );
    pokeb ( 0xA000, LOWORD ( addr ), color );
}

main ()
{
    SetTridentMode ( 0x5D );      // 640x480x256

    for ( int i = 0; i < 640; i++ )
            for ( int j = 0; j < 480; j++ )
                    WritePixel ( i, j, ((i/20)+1)*(j/20+1) );

    getch ();
}
```

Аналогичный пример для SVGA Cirrus Logic выглядит следующим образом:

```
// File Cirrus.cpp
// test for Cirrus Logic 52xx cards
#include  <conio.h>
#include  <dos.h>
#include  <process.h>
#include  <stdio.h>

#define LOWORD(l)          ((int)(l))
#define HIWORD(l)          ((int)((l) >> 16))

inline    void   WriteReg ( int base, int reg, int value )
{
    outportb ( base,     reg );
    outportb ( base + 1, value );
}

inline    char   ReadReg ( int base, int reg )
{
    outportb ( base, reg );

    return inportb ( base + 1 );
}

static    int    CurBank = 0;

// check bits specified by mask in port for being readable/writable
int TestPort ( int port, char mask )
{
    char   save = inportb ( port );

    outportb ( port, save & ~mask );

    char   v1 = inportb ( port ) & mask;

    outportb ( port, save | mask );

    char   v2 = inportb ( port ) & mask;

    outportb ( port, save );

    return v1 == 0 && v2 == mask;
```

```
}

int TestReg ( int port, int reg, char mask )
{
    outportb ( port, reg );

    return TestPort ( port + 1, mask );
}

int FindCirrus ()
{
    char    save = ReadReg ( 0x3C4, 6 );
    int     res  = 0;

    WriteReg ( 0x3C4, 6, 0x12 );        // enable extended registers

    if ( ReadReg ( 0x3C4, 6 ) == 0x12 )
            if ( TestReg ( 0x3C4, 0x1E, 0x3F ) && TestReg ( 0x3D4, 0x1B, 0xFF ) )
                    res = 1;

    WriteReg ( 0x3C4, 6, save );

    return res;
}

void        SetCirrusMode ( int mode )
{
    asm {
            mov     ax, mode
            int     10h

            mov     dx, 3C4h        // enable extended registers
            mov     al, 6
            out     dx, al
            inc     dx
            mov     al, 12h
            out     dx, al
    }
}

void        SetCirrusBank ( int start )
{
    if ( start == CurBank )
            return;

    CurBank = start;
    asm {
            mov     dx, 3CEh
            mov     al, 9
            mov     ah, byte ptr start
            mov     cl, 4
            shl     ah, cl
            out     dx, ax
    }
}

void        WritePixel ( int x, int y, int color )
{
    long    addr = 640l * (long)y + (long)x;

    SetCirrusBank ( HIWORD ( addr ) );
    pokeb ( 0xA000, LOWORD ( addr ), color );
}

main ()
{
```

```
    if ( !FindCirrus () )
    {
        printf ( "\nCirrus card not found" );
        exit ( 1 );
    }

    SetCirrusMode ( 0x5F );      // 640x480x256

    for ( int i = 0; i < 640; i++ )
        for ( int j = 0; j < 480; j++ )
            WritePixel ( i, j, ((i/20)+1)*(j/20+1) );

    getch ();
}
```

Тем самым можно построить библиотеку, обеспечивающую работу с основными SVGA-картами. Сильная привязанность подобной библиотеки к конкретному набору карт - это ее главный недостаток.

Ассоциацией стандартов в области видеоэлектроники VESA (Video Electronic Standarts Association) была сделана попытка стандартизации работы с различными SVGA-платами путем добавления BIOS-платы некоторого стандартного набора функций, обеспечивающего получение необходимой информации о карте, установку заданного режима и банка памяти. При этом также вводится стандартный набор расширенных режимов. Номер режима является 16-битовым шелом, где биты 9-15 зарезервированы и должны быть равны 0, бит 8 для VESA-режимов равен 1, а для родных режимов карты равен 0. Приведем таблицу основных VESA-режимов.

Номер	Разрешение	Бит на пиксел	Количество цветов
100h	640x400	8	256
101h	640x480	8	256
102h	800x600	4	16
103h	800x600	8	256
104h	1024x768	4	16
105h	1024x768	8	256
106h	1280x 1024	4	16
107h	1280x1024	8	256
10Dh	320x200	15	32K
10Eh	320x200	16	64K
10Fh	320x200	24	16M
110h	640x480	15	32K
111h	640x480	16	64K
112h	640x480	24	16M
113h	800x600	15	32K
114h	800x600	16	64K
115h	800x600	24	16M
116h	1024x768	15	32K
117h	1024x768	16	64K
118h	1024x768	24	16M
119h	1280x1024	15	32K
11Ah	1280x1024	16	64K
11Bh	1280x1024	24	16M

Ниже приводятся файлы, содержащие необходимые структуры и функции для работы с VESA-совместимыми адаптерами.

```
// File Vesa.h
#ifndef  __VESA__
#define  __VESA__
                          // 256-color modes
#define  VESA_640x400x256  0x100
#define  VESA_640x480x256  0x101
#define  VESA_800x600x256  0x103
#define  VESA_1024x768x256 0x105
#define  VESA_1280x1024x256 0x107
                          // 32K color modes
#define  VESA_320x200x32K  0x10D
#define  VESA_640x480x32K  0x110
#define  VESA_800x600x32K  0x113
#define  VESA_1024x768x32K 0x116
#define  VESA_1280x1024x32K 0x119
                          // 64K color modes
#define  VESA_320x200x64K  0x10E
#define  VESA_640x480x64K  0x111
#define  VESA_800x600x64K  0x114
#define  VESA_1024x768x64K 0x117
#define  VESA_1280x1024x64K 0x11A
                          // 16M color mode
#define  VESA_320x200x16M  0x10F
#define  VESA_640x480x16M  0x112
#define  VESA_800x600x16M  0x115
#define  VESA_1024x768x16M 0x118
#define  VESA_1280x1024x16M       0x11B

struct   VESAInfo
{
   char      Sign [4]; // 'VESA' signature
   int       Version;  // VESA BIOS version
   char far * OEM;          // Original Equipment Manufactureer string
   long      Capabilities;
   int far   * ModeList;    // list of supported modes
   int       TotalMemory;   // total memory on board in 64Kb blocks
   char      Reserved [236];
};

struct   VESAModeInfo
{
   int       ModeAttributes;
   char      WinAAttributes;
   char      WinBAttributes;
   int       WinGranularity;
   int       WinSize;
   unsigned   WinASegment;
   unsigned   WinBSegement;
   void far * WinFuncPtr;
   int       BytesPerScanLine;
                   // optional data
   int       XResolution;
   int       YResolution;
   char      XCharSize;
   char      YCharSize;
   char      NumberOfPlanes;
   char      BitsPerPixel;
   char      NumberOfBanks;
   char      MemoryModel;
   char      BankSize;
   char      NumberOfPages;
   char      Reserved;
                   // direct color fields
   char      RedMaskSize;
   char      RedFieldPosition;
```

```
   char     GreenMaskSize;
   char     GreenFieldPosition;
   char     BlueMaskSize;
   char     BlueFieldPosition;
   char     RsvdMaskSize;
   char     RsvdFieldPosition;
   char     DirectColorModeInfo;
   char     Resererved2 [216];
};

int      FindVESA ( VESAInfo& );
int      FindVESAMode ( int, VESAModeInfo& );
int      SetVESAMode ( int );
int      GetVESAMode ();
void     SetVESABank ();
#endif
```

```
// File Vesa.cpp
#include <conio.h>
#include <dos.h>
#include <process.h>
#include <stdio.h>
#include <string.h>
#include "Vesa.h"

#define LOWORD(l)           ((int)(l))
#define HIWORD(l)           ((int)((l) >> 16))

inline   int   RGBColor ( int red, int green, int blue )
{
   return ( ( red >> 3 ) << 10 ) | ( ( green >> 3 ) << 5 ) | ( blue >> 3 );
}

static   int         CurBank     = 0;
static   int         Granularity = 1;
static   VESAModeInfoCurMode;

int      FindVESA ( VESAInfo& vi )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
   asm {
         push  es
         push  di
         les   di, dword ptr vi
         mov   ax, 4F00h
         int   10h
         pop   di
         pop   es
   }
#else
   asm {
         push  di
         mov   di, word ptr vi
         mov   ax, 4F00h
         int   10h
         pop   di
   }
#endif
   if ( _AX != 0x004F )
         return 0;

   return !strncmp ( vi.Sign, "VESA", 4 );
}

int      FindVESAMode ( int mode, VESAModeInfo& mi )
```

```
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
   asm {
         push  es
         push  di
         les   di, dword ptr mi
         mov   ax, 4F01h
         mov   cx, mode
         int   10h
         pop   di
         pop   es
   }
#else
   asm {
         push  di
         mov   di, word ptr mi
         mov   ax, 4F01h
         mov   cx, mode
         int   10h
         pop   di
   }
#endif
   return _AX == 0x004F;

}

int      SetVESAMode ( int mode )
{
   if ( !FindVESAMode ( mode, CurMode ) )
         return 0;

   Granularity = 64 / CurMode.WinGranularity;
   asm {
         mov   ax, 4F02h
         mov   bx, mode
         int   10h
   }

   return _AX == 0x004F;
}

int      GetVESAMode ()
{
   asm {
         mov   ax, 4F03h
         int   10h
   }

   if ( _AX != 0x004F )
         return 0;
   else
         return _BX;
}

void     SetVESABank ( int start )
{
   if ( start == CurBank )
         return;

   CurBank = start;
   start  *= Granularity;
   asm {
         mov   ax, 4F05h
         mov   bx, 0
         mov   dx, start
         push  dx
```

```
            int   10h
            mov   bx, 1
            pop   dx
            int   10h
      }
   }

   void     WritePixel ( int x, int y, int color )
   {
      long  addr = (long)CurMode.BytesPerScanLine * (long)y + (long)(x<<1);

      SetVESABank ( HIWORD ( addr ) );
      poke ( 0xA000, LOWORD ( addr ), color );
   }

   main ()
   {
      VESAInfo Info;

      if ( !FindVESA ( Info ) )
      {
            printf ( "VESA VBE not found" );
            exit ( 1 );
      }

      if ( !SetVESAMode ( VESA_640x480x32K ) )
      {
            printf ( "Mode not supported" );
            exit ( 1 );
      }

      for ( int i = 0; i < 256; i++ )
            for ( int j = 0; j < 256; j++ )
            {
                  WritePixel ( 320-i, 240-j, RGBColor ( 0,j,i) );
                  WritePixel ( 320+i, 240-j, RGBColor ( i,j,i) );
                  WritePixel ( 320+i, 240+j, RGBColor ( j,i,i) );
                  WritePixel ( 320-i, 240+j, RGBColor ( j,0,i) );
            }

      getch ();
   }
```

При помощи функции FindVESA можно получить информацию о наличии VESA BIOS, а также узнать все режимы, доступные для данной карты.

Функция FindVESAMode возвращает информацию о режиме в полях структуры VESAModeInfo.

Укажем наиболее важные поля.

Поле	*Размер в байтах*	*Комментарий*
ModeAttributes	2	Характеристики режима: бит 0 - режим доступен, бит 1 - режим зарезервирован, бит 2 - BIOS поддерживает вывод в этом режиме, бит 3 - режим цветной, бит 4 - режим графический
WinAAttributes	1	Характеристики банка A: бит 0 - банк поддерживается, бит 1 - из банка можно читать, бит 2 - в банк можно писать
WinBAttributes	1	Характеристики банка B
WinGranularity	2	Шаг установки банка в килобайтах
WinSize	2	Размер банка
WinASegment	2	Сегментный адрес банка A
WinBSegment	2	Сегментный адрес банка B

BytesPerScanLine	2	Количество байт под одну строку
BitsPerPixel	1	Количество бит, отводимых под один пиксел
NumberOfBanks	1	Количество банков памяти

Приведем программу, выдающую информацию по всем доступным VESA-режимам.

```
// File VesaInfo.cpp
char * ColorInfo ( int bits )
{
   switch ( bits )
   {
         case 4:     return "16 colors";
         case 8:     return "256 colors";
         case 15:    return "32K colors ( HiColor )";
         case 16:    return "64K colors ( HiColor )";
         case 24:    return "16M colors ( TrueColor )";
         default:    return "";
   }
}

void     DumpMode ( int mode )
{
   VESAModeInfo mi;

   if ( !FindVESAMode ( mode, mi ) )
         return;

   if ( ( mi.ModeAttributes & 1 ) == 0 )      // not available now
         return;

   printf ( "\n%4X %10s %4dx%4d %2d %s", mode,
               mi.ModeAttributes & 0x10 ? "Graphics" : "Text",
               mi.XResolution, mi.YResolution, mi.BitsPerPixel,
               ColorInfo ( mi.BitsPerPixel ) );
}

main ()
{
   VESAInfo Info;
   char   str [256];

   if ( !FindVESA ( Info ) )
   {
         printf ( "VESA VBE not found" );
         exit ( 1 );
   }

        _fstrcpy ( str, Info.OEM );
   printf ( "\nVESA VBE version %d.%d\nOEM: %s\nTotal memory: %dKb\n",
          Info.Version >> 8, Info.Version & 0xFF, str, Info.TotalMemory * 64 );

   for ( int i = 0; Info.ModeList [i] != -1; i++ )
         DumpMode ( Info.ModeList [i] );
}
```

Непалитровые режимы адаптеров SVGA

Ряд SVGA-карт поддерживают использование так называемых непалитровых режимов - для каждого пиксела вместо индекса в палитре непосредственно задается его RGB-значение.

Обычно такими режимами являются режимы HiColor (15 или 16 бит на пиксел) и TrueColor (24 бита на пиксел).

Видеопамять для этих режимов устроена аналогично 256-цветным режимам SVGA - под каждый пиксел отводится целое количество байт памяти (2 байта для HiColor и 3 байта для TrueColor), и все они расположены подряд и сгруппированы в банки.

Наиболее простой является организация режима TrueColor (16 миллионов цветов) - под каждую из трех компонент цвета отводится по одному байту.

Несколько сложнее организация режимов HiColor, где под каждый пиксел отводится по 2 байта и возможны два варианта:

- под каждую компоненту отводится по 5 бит, последний бит не используется (32 тысячи цветов);
- под красную и синюю компоненты отводится по 5 бит, под зеленую - 6 бит (64 тысячи цветов).

Ниже приводится простая программа, иллюстрирующая работу с режимом HiColor 32 тысячи цветов.

```
// File HiColor.cpp
#include <conio.h>
#include <dos.h>
#include <process.h>
#include <stdio.h>
#include <string.h>
#include "Vesa.h"

#define LOWORD(l)           ((int)(l))
#define HIWORD(l)           ((int)((l) >> 16))

inline   int   RGBColor ( int red, int green, int blue )
{
   return ( ( red >> 3 ) << 10 ) | ( ( green >> 3 ) << 5 ) | ( blue >> 3 );
}

static   int          CurBank     = 0;
static   int          Granularity = 1;
static   VESAModeInfoCurMode;

int      FindVESA ( VESAInfo& vi )
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
   asm {
         push  es
         push  di
         les   di, dword ptr vi
         mov   ax, 4F00h
         int   10h
         pop   di
         pop   es
   }
#else
   asm {
         push  di
         mov   di, word ptr vi
         mov   ax, 4F00h
         int   10h
         pop   di
   }
#endif
   if ( _AX != 0x004F )
         return 0;

   return !strncmp ( vi.Sign, "VESA", 4 );
}

int      FindVESAMode ( int mode, VESAModeInfo& mi )
```

```
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
   asm {
         push  es
         push  di
         les   di, dword ptr mi
         mov   ax, 4F01h
         mov   cx, mode
         int   10h
         pop   di
         pop   es
   }
#else
   asm {
         push  di
         mov   di, word ptr mi
         mov   ax, 4F01h
         mov   cx, mode
         int   10h
         pop   di
   }
#endif
   return _AX == 0x004F;

}

int      SetVESAMode ( int mode )
{
   if ( !FindVESAMode ( mode, CurMode ) )
         return 0;

   Granularity = 64 / CurMode.WinGranularity;
   asm {
         mov   ax, 4F02h
         mov   bx, mode
         int   10h
   }

   return _AX == 0x004F;
}

int      GetVESAMode ()
{
   asm {
         mov   ax, 4F03h
         int   10h
   }

   if ( _AX != 0x004F )
         return 0;
   else
         return _BX;
}

void     SetVESABank ( int start )
{
   if ( start == CurBank )
         return;

   CurBank = start;
   start  *= Granularity;
   asm {
         mov   ax, 4F05h
         mov   bx, 0
         mov   dx, start
         push  dx
```

```
            int     10h
            mov     bx, 1
            pop     dx
            int     10h
      }
}

void      WritePixel ( int x, int y, int color )
{
      long  addr = (long)CurMode.BytesPerScanLine * (long)y + (long)(x<<1);

      SetVESABank ( HIWORD ( addr ) );
      poke ( 0xA000, LOWORD ( addr ), color );
}

main ()
{
      VESAInfo Info;

      if ( !FindVESA ( Info ) )
      {
            printf ( "VESA VBE not found" );
            exit ( 1 );
      }

      if ( !SetVESAMode ( VESA_640x480x32K ) )
      {
            printf ( "Mode not supported" );
            exit ( 1 );
      }

      for ( int i = 0; i < 256; i++ )
            for ( int j = 0; j < 256; j++ )
            {
                  WritePixel ( 320-i, 240-j, RGBColor ( 0,j,i) );
                  WritePixel ( 320+i, 240-j, RGBColor ( i,j,i) );
                  WritePixel ( 320+i, 240+j, RGBColor ( j,i,i) );
                  WritePixel ( 320-i, 240+j, RGBColor ( j,0,i) );
            }

      getch ();
}
```

ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ И В ПРОСТРАНСТВЕ

Хотя время и причисляют к непрерывным величинам, однако оно, будучи незримым и без тела, не целиком подпадает власти геометрии, <...> точка во времени должна быть приравнена к мгновению, а линия имеет сходство с длительностью известного количества времени <...>, и если линия делима до бесконечности, то и промежуток времени не чужд такого деления.

Леонардо да Винчи

Вывод изображения на экран дисплея и разнообразные действия с ним, в том числе и визуальный анализ, требуют от пользователя известной геометрической грамотности. Геометрические понятия, формулы и факты, относящиеся прежде всего к плоскому и трехмерному случаям, играют в задачах компьютерной графики особую роль. Геометрические соображения, подходы и идеи в соединении с постоянно расширяющимися возможностями вычислительной техники являются неиссякаемым источником существенных продвижений на пути развития компьютерной графики, ее эффективного использования в научных и иных исследованиях. Порой даже самые простые геометрические методики обеспечивают заметные продвижения на отдельных этапах решения большой графической задачи. С простых геометрических рассмотрении мы и начнем наш рассказ. Заметим прежде всего, что особенности использования геометрических понятий, формул и фактов, как простых и хорошо известных, так и новых более сложных, требуют особого взгляда на них и иного осмысления.

Аффинные преобразования на плоскости

В компьютерной графике все, что относится к двумерному случаю, принято обозначать символом (2D) (2-dimension).

Допустим, на плоскости введена прямолинейная координатная система. Тогда каждой точке М ставится в соответствие упорядоченная пара чисел (x, у) ее координат (рис. 1). Вводя на плоскости еще одну прямолинейную систему координат, мы ставим в соответствие той же точке М другую пару чисел - (x^*, y^*).

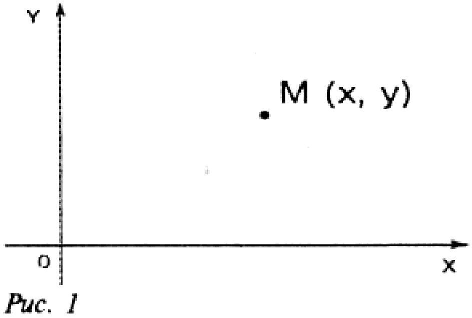


Рис. 1

Переход от одной прямолинейной координатной системы на плоскости к другой описывается следующими соотношениями:

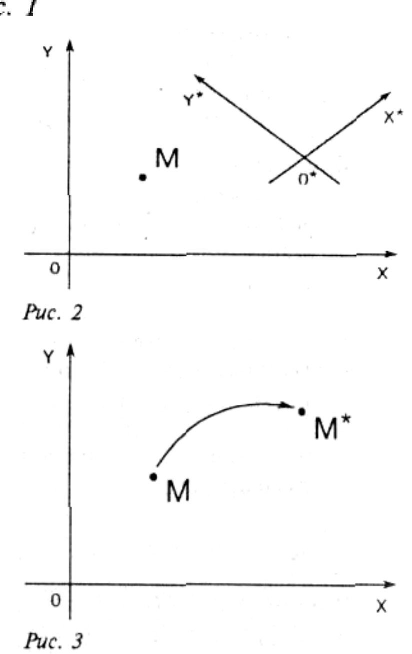
$x^* = ax + by + \lambda.,$

$y^* = yx + by + \lambda\,,$ (*)

где а, В, у, λ, и - произвольные числа, связанные неравенством

$$\begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0.$$





Замечание

Формулы (*) можно рассматривать двояко: либо сохраняется точка и изменяется координатная система (рис. 2) - в этом случае произвольная точка М остается той же, изменяются лишь ее координаты

$(x, y) \mid (x^*, y^*)$,

либо изменяется точка и сохраняется координатная система (рис. 3) - в этом случае формулы (*) задают отображение, переводящее произвольную точку M(x, y) в точку M(x *, y *), координаты которой определены в той же координатной системе.*

В дальнейшем мы будем рассматривать формулы (*) как правило, согласно которому в заданной системе прямолинейных координат преобразуются точки плоскости.

В аффинных преобразованиях плоскости особую роль играют несколько важных частных случаев, имеющих хорошо прослеживаемые геометрические характеристики. При исследовании геометрического смысла числовых коэффициентов в формулах (*) для этих случаев нам удобно считать, что заданная система координат является прямоугольной декартовой.

А. Поворот (вокруг начальной точки на угол φ) (рис. 4) описывается формулами

$x^* = x \cos\varphi - y \sin\varphi$,

$y^* = x \sin\varphi + y \cos\varphi$.

Рис. 4

Б. Растяжение (сжатие) вдоль координатных осей можно задать так:

$x^* = ax$, $y^* = 5y$,

$a > 0$, $5 > 0$.

Растяжение (сжатие) вдоль оси абсцисс обеспечивается при условии, что $a > 1$ ($a < 1$) На рис. 5 $a = 8 > 1$.

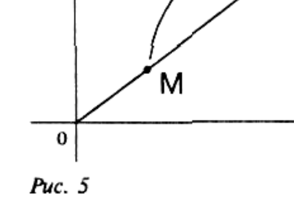


Рис. 5

В. Отражение (относительно оси абсцисс) (рис. 6) задается при помощи формул

$x^* = x$,

$y^* = -y$.

Г. На рис. 7 вектор переноса MM* имеет координаты λ и μ. Перенос обеспечивают соотношения

$X^* = x + \lambda$,

$y^* = y + \mu$.

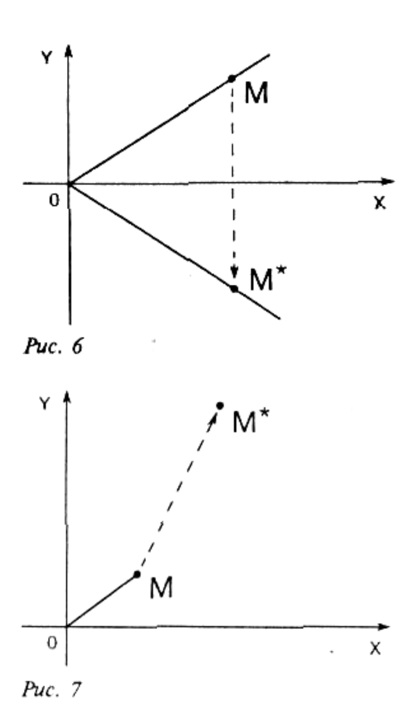


Рис. 6

Рис. 7

Выбор этих четырех частных случаев определяется двумя обстоятельствами.

1. Каждое из приведенных выше преобразований имеет простой и наглядный геометрический смысл (геометрическим смыслом наделены и постоянные числа, входящие в приведенные формулы).

2. Как доказывается в курсе аналитической геометрии, любое преобразование вида (*) всегда можно представить как последовательное исполнение (суперпозицию) простейших преобразований вида А, Б, В и Г (или части этих преобразований).

Таким образом, справедливо следующее важное свойство аффинных преобразований плоскости: любое отображение вида (*) можно описать при помощи отображений, задаваемых формулами А, Б, В и Г.

Для эффективного использования этих известных формул в задачах компьютерной графики более удобной является их матричная запись. Матрицы, соответствующие случаям А, Б и В, строятся легко и имеют соответственно следующий вид:

$$\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}, \begin{pmatrix} \alpha & 0 \\ 0 & \delta \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Однако для решения рассматриваемых далее задач весьма желательно охватить матричным подходом все четыре простейших преобразования (в том числе и перенос), а, значит, и общее аффинное преобразование. Этого можно достичь, например, так: перейти к описанию произвольной точки плоскости не упорядоченной парой чисел, как это было сделано выше, а упорядоченной тройкой чисел.

Однородные координаты точки

Пусть М - произвольная точка плоскости с координатами x и y, вычисленными относительно заданной прямолинейной координатной системы. Однородными координатами этой точки называется любая тройка одновременно неравных нулю чисел x_1, x_2, x_3, связанных с заданными числами x и y следующими соотношениями:

$$\frac{x_1}{x_3} = x\,, \ \frac{x_2}{x_3} = y\,.$$

При решении задач компьютерной графики однородные координаты обычно вводятся так: произвольной точке M(x, y) плоскости ставится в соответствие точка M∈ (x, y, 1) в пространстве (рис. 8).

Заметим, что произвольная точка на прямой, соединяющей начало координат, точку O(0, 0, 0), с точкой M∈ (x, y, 1), может быть задана тройкой чисел вида

(hx, hy, h).

Будем считать, что $h \neq 0$.

Вектор с координатами hx, hy, h является направляющим вектором прямой, соединяющей точки 0(0, 0, 0) и M∈(x, y, 1). Эта прямая пересекает плоскость z = 1 в точке (x, y, 1), которая однозначно определяет точку (x, y) координатной плоскости xy. Тем самым между произвольной точкой с координатами (x, y) и множеством троек чисел вида

(hx, hy, h), $h \neq 0$,

устанавливается (взаимно однозначное) соответствие, позволяющее считать числа hx, hy, h новыми координатами этой точки.

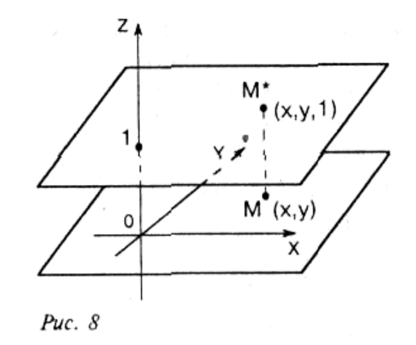


Рис. 8

Замечание

Широко используемые в проективной геометрии однородные координаты позволяют эффективно описывать так называемые несобственные элементы (по существу, те, которыми проективная плоскость отличается от привычной нам евклидовой плоскости). Более подробно о новых возможностях, предоставляемых введенными одно родными координатами, говорится в четвертом разделе этой главы.

В проективной геометрии для однородных координат принято следующее обозначение:

$x : y : 1$

или, более общо, $x_1 : x_2 : x_3$

(напомним, что здесь непременно требуется, чтобы числа x_1, x_2, x_3 одновременно в нуль не обращались).

Применение однородных координат оказывается удобным уже При решении простейших задач.

Рассмотрим, например, вопросы, связанные с изменением масштаба. Если устройство отображения работает только с целыми числами (или если необходимо работать только с целыми числами), то для произвольного значения h (например, h = 1) точку с однородными координатами (0.5 0.1 2.5) представить нельзя. Однако при разумном выборе h можно добиться того, чтобы координаты этой точки были целыми числами. В частности, при h = 10 для рассматриваемого примера имеем

(5 1 25).

Рассмотрим другой случай. Чтобы результаты преобразования не приводили к арифметическому переполнению, для точки с координатами

(80000 40000 1000)

можно взять, например, h = 0,001. В результате получим (80 40 1).

Приведенные примеры показывают полезность использования однородных координат при проведении расчетов. Однако основной целью введения однородных координат в компьютерной графике является их несомненное удобство в применении к геометрическим преобразованиям.

При помощи троек однородных координат и матриц третьего порядка можно описать любое аффинное преобразование плоскости.

В самом деле, считая h = 1, сравним две записи: помеченную символом * и нижеследующую, матричную:

$$\begin{pmatrix} x^* & y^* & 1 \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{bmatrix} \alpha & \gamma & 0 \\ \beta & \delta & 0 \\ \lambda & \mu & 1 \end{bmatrix}$$

Нетрудно заметить, что после перемножения выражений, стоящих в правой части последнего соотношения, мы получим обе формулы (*) и верное числовое равенство $1 = 1$.

Тем самым сравниваемые записи можно считать равносильными.

Замечание

Иногда в литературе используется другая запись - запись по столбцам:

$$\begin{bmatrix} x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & \lambda \\ \gamma & \delta & \mu \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Такая запись эквивалентна приведенной выше записи по строкам (и получается из нее транспонированием).

Элементы произвольной матрицы аффинного преобразования не несут в себе явно выраженного геометрического смысла. Поэтому чтобы реализовать то или иное отображение, то есть найти элементы соответствующей матрицы по заданному геометрическому описанию, необходимы специальные приемы. Обычно построение этой матрицы в соответствии со сложностью рассматриваемой задачи и с описанными выше частными случаями разбивают на несколько этапов.

На каждом этапе ищется матрица, соответствующая тому или иному из выделенных выше случаев А, Б, В или Г, обладающих хорошо выраженными геометрическими свойствами.

Выпишем соответствующие матрицы третьего порядка.

А. Матрица вращения (rotation)

$$[R] = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Б. Матрица растяжения(сжатия) (dilatation)

$$[D]=\begin{bmatrix}\alpha & 0 & 0\\ 0 & \delta & 0\\ 0 & 0 & 1\end{bmatrix}$$

В. Матрица отражения (reflection)

$$[M]=\begin{bmatrix}1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Г. Матрица переноса (translation)

$$[T]=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ \lambda & \mu & 1\end{bmatrix}$$

Рассмотрим примеры аффинных преобразований плоскости.

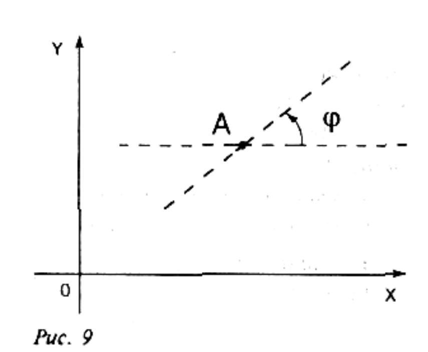
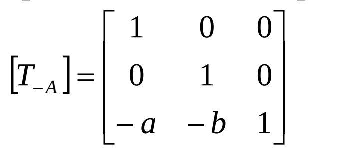


Рис. 9

Пример 1

Построить матрицу поворота вокруг точки A(a, b) на угол φ (рис. 9).

1-й шаг. Перенос на вектор -A(-a, -b) для совмещения центра поворота с началом координат;



$$[T_{-A}]=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ -a & -b & 1\end{bmatrix}$$

матрица соответствующего преобразования.

2-й шаг. Поворот на угол φ;

$$[R_\varphi]=\begin{bmatrix}\cos\varphi & \sin\varphi & 0\\ -\sin\varphi & \cos\varphi & 0\\ 0 & 0 & 1\end{bmatrix}$$

матрица соответствующего преобразования.

3-й шаг. Перенос на вектор A(a, b*)* для возвращения центра поворота в прежнее положение;

$$[T_A]=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{bmatrix}$$

матрица соответствующего преобразования.

Перемножим матрицы в том же порядке, как они выписаны:

$[T_{-A}][R_\varphi][T_A]$.

В результате получим, что искомое преобразование (в матричной записи) будет выглядеть следующим образом:

$$(x^* \quad y^* \quad 1)=(x \quad y \quad 1)\times\begin{bmatrix}\cos\varphi & \sin\varphi & 0\\ -\sin\varphi & \cos\varphi & 0\\ -a\cos\varphi+b\sin\varphi+a & -a\sin\varphi-b\cos\varphi+b & 1\end{bmatrix}$$

Элементы полученной матрицы (особенно в последней строке) не так легко запомнить. В то же время каждая из трех перемножаемых матриц по геометрическому описанию соответствующего отображения легко строится.

Пример 2

Построить матрицу растяжения с коэффициентами растяжения α вдоль оси абсцисс и β вдоль оси ординат и с центром в точке A(a, b).

1-й шаг. Перенос на вектор -A(-a, -b) для совмещения центра растяжения с началом координат;

$$[T_{-A}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{bmatrix}$$

матрица соответствующего преобразования.

2-й шаг. Растяжение вдоль координатных осей с коэффициентами a и 5 соответственно; матрица преобразования имеет вид

$$[D] = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3-й шаг. Перенос на вектор A(a, b) для возвращения центра растяжения в прежнее положение; матрица соответствующего преобразования -

$$[T_A] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

Перемножив матрицы в том же порядке

$[T_{-A}][D][T_A]$,

получим окончательно

$$\begin{pmatrix} x^* & y^* & 1 \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ (1-\alpha)a & (1-\delta)b & 1 \end{bmatrix}$$

Замечание

Рассуждая подобным образом, то есть разбивая предложенное преобразование на этапы, поддерживаемые матрицами

[R],[D],[M],[T],

можно построить матрицу любого аффинного преобразования по его геометрическому описанию.

Аффинные преобразования в пространстве

Обратимся теперь к трехмерному случаю (3D) (3-dimension) и начнем наши рассмотрения сразу с введения однородных координат.

Поступая аналогично тому, как это было сделано в размерности два, заменим координатную тройку (x, y, z), задающую точку в пространстве, на четверку чисел

(x y z 1) или, более общо, на четверку

(hx hy hz), $h \neq 0$.

Каждая точка пространства (кроме начальной точки О) может быть задана четверкой одновременно не равных нулю чисел; эта четверка чисел определена однозначно с точностью до общего множителя.

Предложенный переход к новому способу задания точек дает возможность воспользоваться матричной записью и в более сложных, трехмерных задачах.

Любое аффинное преобразование в трехмерном пространстве может быть представлено в виде суперпозиции вращений, растяжений, отражений и переносов. Поэтому вполне уместно

сначала подробно описать матрицы именно этих преобразований (ясно, что в данном случае порядок матриц должен быть равен четырем).

А. Матрицы вращения в пространстве

Матрица вращения вокруг оси абсцисс на угол φ:

$$[R_x]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\varphi & \sin\varphi & 0\\ 0 & -\sin\varphi & \cos\varphi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Матрица вращения вокруг оси ординат на угол ψ:

$$[R_y]=\begin{bmatrix}\cos\psi & 0 & -\sin\psi & 0\\ 0 & 1 & 0 & 0\\ \sin\psi & 0 & \cos\psi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Матрица вращения вокруг оси аппликат на угол χ:

$$[R_y]=\begin{bmatrix}\cos\chi & \sin\chi & 0 & 0\\ -\sin\chi & \cos\chi & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Замечание

Полезно обратить внимание на место знака "-" в каждой из трех приведенных матриц.

Б. Матрица растяжения (сжатия):

$$[D]=\begin{bmatrix}\alpha & 0 & 0 & 0\\ 0 & \beta & 0 & 0\\ 0 & 0 & \gamma & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

где

$\alpha > 0$ - коэффициент растяжения (сжатия) вдоль оси абсцисс;

$\beta > 0$ - коэффициент растяжения (сжатия) вдоль оси ординат;

$\gamma > 0$ - коэффициент растяжения (сжатия) вдоль оси аппликат).

В. Матрицы отражения

Матрица отражения относительно плоскости *ху*:

$$[M_z]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & -1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Матрица отражения относительно плоскости *yz:*

$$[M_x]=\begin{bmatrix}-1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Матрица отражения относительно плоскости *zx*:

$$[M_y]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & -1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Г. Матрица переноса (здесь (λ, μ , ν) - вектор переноса):

$$[T]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ \lambda & \mu & \nu & 1\end{bmatrix}$$

Замечание

Как и в двумерном случае, все выписанные матрицы невырождены.

Приведем важный пример построения матрицы сложного преобразования по его геометрическому описанию.

Пример 1

Построить матрицу вращения на угол φ вокруг прямой L, проходящей через точку A(a, b, c) и имеющую направляющий вектор (l, m, n). Можно считать, что направляющий вектор прямой является единичным:

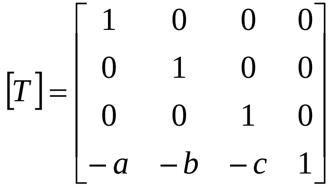
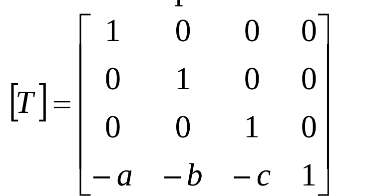
$l^2 + m^2 + n^2 = 1$

На рис. 10 схематично показано, матрицу какого преобразования требуется найти.

Рис. 10

Решение сформулированной задачи разбивается на несколько шагов. Опишем последовательно каждый из них.

1-й шаг. Перенос на вектор -A(-a, -b, -c) при помощи матрицы

$$[T]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ -a & -b & -c & 1\end{bmatrix}$$

В результате этого переноса мы добиваемся того, чтобы прямая L проходила через начало координат.

2-й шаг. Совмещение оси аппликат с прямой L двумя поворотами вокруг оси абсцисс и оси ординат.

1-й поворот - вокруг оси абсцисс на угол ψ (подлежащий определению). Чтобы найти этот угол, рассмотрим ортогональную проекцию L' исходной прямой L на плоскость X = 0 (рис. 11).

Направляющий вектор прямой L' определяется просто - он равен

(0, m, n).

Отсюда сразу же вытекает, что

$$\cos\psi = \frac{n}{d}, \sin\psi = \frac{m}{d},$$

где

$$d = \sqrt{m^2 + n^2}.$$

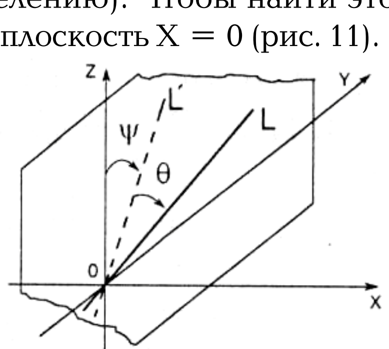


Рис. 11

Соответствующая матрица вращения имеет следующий вид:

$$[R_x]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \frac{n}{d} & \frac{m}{d} & 0\\ 0 & -\frac{m}{d} & \frac{n}{d} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Под действием преобразования, описываемого этой матрицей, координаты вектора (l, m, n) изменятся. Подсчитав их, в результате получим

$$(l,m,n,1)[R_x]=(l,0,d,1).$$

2-й поворот - вокруг оси ординат на угол θ, определяемый соотношениями

$\cos\theta = l$, $\sin\theta = -d$.

Соответствующая матрица вращения записывается в следующем виде:

$$[R_y]=\begin{bmatrix}l & 0 & d & 0\\ 0 & 1 & 0 & 0\\ -d & 0 & l & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

3-й шаг. Вращение вокруг прямой L на заданный угол φ. Так как теперь прямая L совпадает с осью аппликат, то соответствующая матрица имеет следующий вид:

$$[R_z]=\begin{bmatrix}\cos\varphi & \sin\varphi & 0 & 0\\ -\sin\varphi & \cos\varphi & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

4-й шаг. Поворот вокруг оси ординат на угол -θ.

5-й шаг. Поворот вокруг оси абсцисс на угол -ψ.

Замечание

Вращение в пространстве некоммутативно. Поэтому порядок, в котором проводятся вращения, является весьма существенным.

6-й шаг. Перенос на вектор А*(a, b, c).*

Перемножив найденные матрицы в порядке их построения, получим следующую матрицу:

$$[T][R_x][R_y][R_z][R_y]^{-1}[R_x]^{-1}[T]^{-1}.$$

Выпишем окончательный результат, считая для простоты, что ось вращения L проходит через начальную точку:

$$\begin{pmatrix}l^2+\cos\varphi(1-l^2) & l(1-\cos\varphi)m+n\sin\varphi & l(1-\cos\varphi)n-m\sin\varphi & 0\\ l(1-\cos\varphi)m-n\sin\varphi & m^2+\cos\varphi(1-m^2) & m(1-\cos\varphi)n+l\sin\varphi & 0\\ l(1-\cos\varphi)n-m\sin\varphi & m(1-\cos\varphi)n-l\sin\varphi & n^2+\cos\varphi(1-n^2) & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Рассматривая другие примеры подобного рода, мы будем получать в результате невырожденные матрицы вида

$$[A]=\begin{bmatrix}\alpha_1 & \alpha_2 & \alpha_3 & 0\\ \beta_1 & \beta_2 & \beta_3 & 0\\ \gamma_1 & \gamma_2 & \gamma_3 & 0\\ \lambda & \mu & \nu & 1\end{bmatrix}$$

При помощи таких матриц можно преобразовывать любые плоские и пространственные фигуры.

Пример 2

Требуется подвергнуть заданному аффинному преобразованию выпуклый многогранник.

Для этого сначала по геометрическому описанию отображения находим его матрицу [A]. Замечая далее, что произвольный выпуклый многогранник однозначно задается набором всех своих вершин

$V_i(x_i, y_i, z_i)$, i = 1,...,n,

строим матрицу

$$V = \begin{pmatrix} x_1 & y_1 & z_1 & 1 \\ . & . & . & . \\ x_n & y_n & z_n & 1 \end{pmatrix}$$

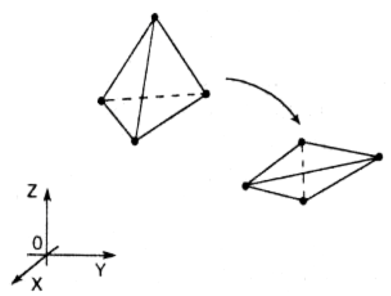


Рис. 12

Подвергая этот набор преобразованию, описываемому найденной невырожденной матрицей четвертого порядка, [V][A], мы получаем набор вершин нового выпуклого многогранника - образа исходного (рис. 12).

Платоновы тела

Правильными многогранниками (платоновыми телами) называются такие выпуклые многогранники, все грани которых суть правильные многоугольники и все многогранные углы при вершинах равны между собой.

Существует ровно пять правильных многогранников (это доказал Евклид). Они - правильный тетраэдр, гексаэдр (куб), октаэдр, додекаэдр и икосаэдр. Их основные характеристики приведены в следующей таблице.

Название многогранника	*Число граней Г*	*Число ребер Р*	*Число вершин В*
Тетраэдр	4	6	4
Гексаэдр	6	12	8
Октаэдр	8	12	6
Додекаэдр	12	30	20
Икосаэдр	20	30	12

Нетрудно заметить, что в каждом из пяти случаев числа Г, Р и В связаны равенством Эйлера Г + В = Р + 2.

Правильные многогранники обладают многими интересными свойствами. Здесь мы коснемся только тех свойств, которые можно применить для построения этих многогранников.

Для полного описания правильного многогранника, вследствие его выпуклости, достаточно указать способ отыскания всех его вершин.

Операции построения первых трех платоновых тел являются особенно простыми. С них и начнем.

Куб (гексаэдр) строится совсем несложно (рис. 13).

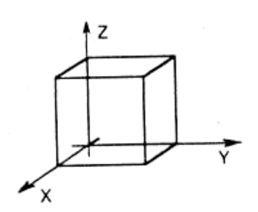
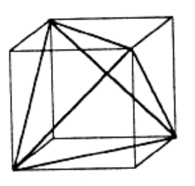


Рис. 13

Покажем, как, используя куб, можно построить тетраэдр и октаэдр.

Для построения тетраэдра достаточно провести скрещивающиеся диагонали противоположных граней куба (рис. 14).

Тем самым вершинами тетраэдра являются любые 4 вершины куба, попарно не смежные ни с одним из его ребер.


Рис. 14

Для построения октаэдра воспользуемся следующим свойством двойственности: вершины октаэдра суть центры (тяжести) граней куба (рис. 15).

И значит, координаты вершин октаэдра по координатам вершин куба легко

вычисляются (каждая координата вершины октаэдра является средним арифметическим одноименных координат четырех вершин содержащей ее грани куба).

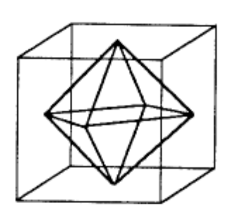


Рис. 15

Додекаэдр и икосаэдр также можно построить при помощи куба. Однако существует, на наш взгляд, более простой способ их конструирования, который мы и собираемся описать здесь.

Начнем с икосаэдра.

Рассечем круглый цилиндр единичного радиуса, ось которого совпадает с осью аппликат Z двумя плоскостями Z=-0.5 и Z=0.5 (рис. 16). Разобьем каждую из полученных окружностей на 5 равных частей так, как показано на рис. 17. Перемещаясь вдоль обеих окружностей против часовой стрелки, занумеруем выделенные 10 точек в порядке возрастания угла поворота (рис. 18) и затем последовательно, в соответствии с нумерацией, соединим эти точки прямолинейными отрезками (рис. 19). Стягивая теперь хордами точки, выделенные на каждой из окружностей, мы получим в результате пояс из 10 правильных треугольников (рис. 20).

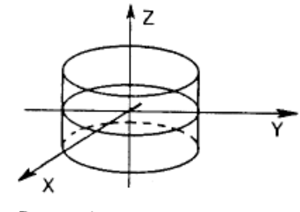


Рис. 16

Рис. 17

Для завершения построения икосаэдра выберем на оси Z две точки так, чтобы длины боковых ребер пятиугольных пирамид с вершинами в этих точках и основаниями, совпадающими с построенными пятиугольниками (рис. 21), были равны длинам сторон пояса из треугольников. Нетрудно видеть, что для этого годятся точки с аппликатами

$$\pm\frac{\sqrt{5}}{2}.$$

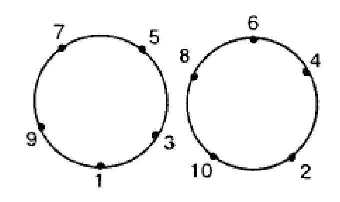


Рис. 18

В результате описанных построений получаем 12 точек. Выпуклый многогранник с вершинами в этих точках будет иметь 20 граней, каждая из которых является правильным треугольником, и все его многогранные углы при вершинах будут равны между собой. Тем самым результат описанного построения - икосаэдр (рис. 22).

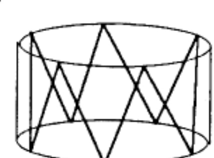
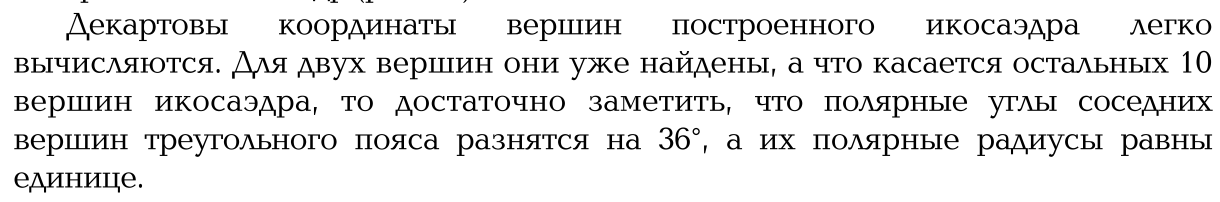


Рис. 19



Декартовы координаты вершин построенного икосаэдра легко вычисляются. Для двух вершин они уже найдены, а что касается остальных 10 вершин икосаэдра, то достаточно заметить, что полярные углы соседних вершин треугольного пояса разнятся на 36°, а их полярные радиусы равны единице.

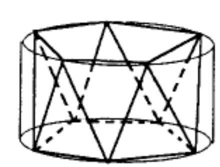


Рис. 20

Остается построить додекаэдр.

Оставляя в стороне способ, предложенный Евклидом (построение "крыш" над гранями куба), вновь воспользуемся свойством двойственности, но теперь уже связывающим додекаэдр и икосаэдр: вершины додекаэдра суть центры (тяжести) треугольных граней икосаэдра.

И значит, координаты каждой вершины додекаэдра можно найти, вычислив средние арифметические соответствующих координат вершин содержащей ее грани икосаэдра (рис. 23).

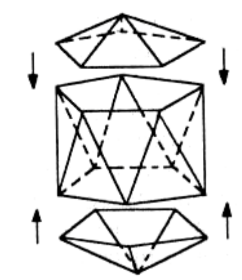


Рис. 21



Замечание

Подвергая полученные правильные многогранники преобразованиям вращения и переноса, можно получить платоновы тела с центрами в произвольных точках и с любыми длинами ребер.

В качестве упражнения полезно написать по предложенным способам программы, генерирующие все платоновы тела.

Рис. 22

Рис. 23

Виды проектирования

Изображение объектов на картинной плоскости связано с еще одной геометрической операцией - проектированием при помощи пучка прямых. В компьютерной графике используется несколько различных видов проектирования (иногда называемого также проецированием). Наиболее употребимые на практике виды проектирования суть параллельное и центральное.

Для получения проекции объекта на картинную плоскость необходимо провести через каждую его точку прямую из заданного проектирующего пучка (собственного или несобственного) и затем найти координаты точки пересечения этой прямой с плоскостью изображения. В случае центрального проектирования все прямые исходят из одной точки - центра собственного пучка. При параллельном проектировании центр (несобственного) пучка считается лежащим в бесконечности (рис. 24).

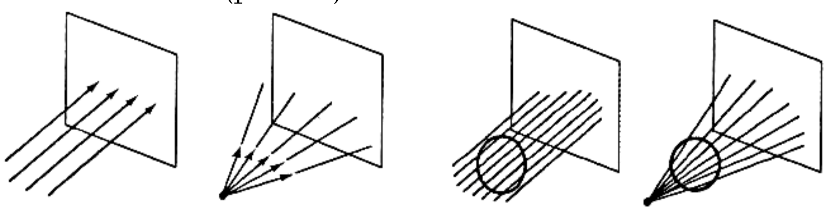
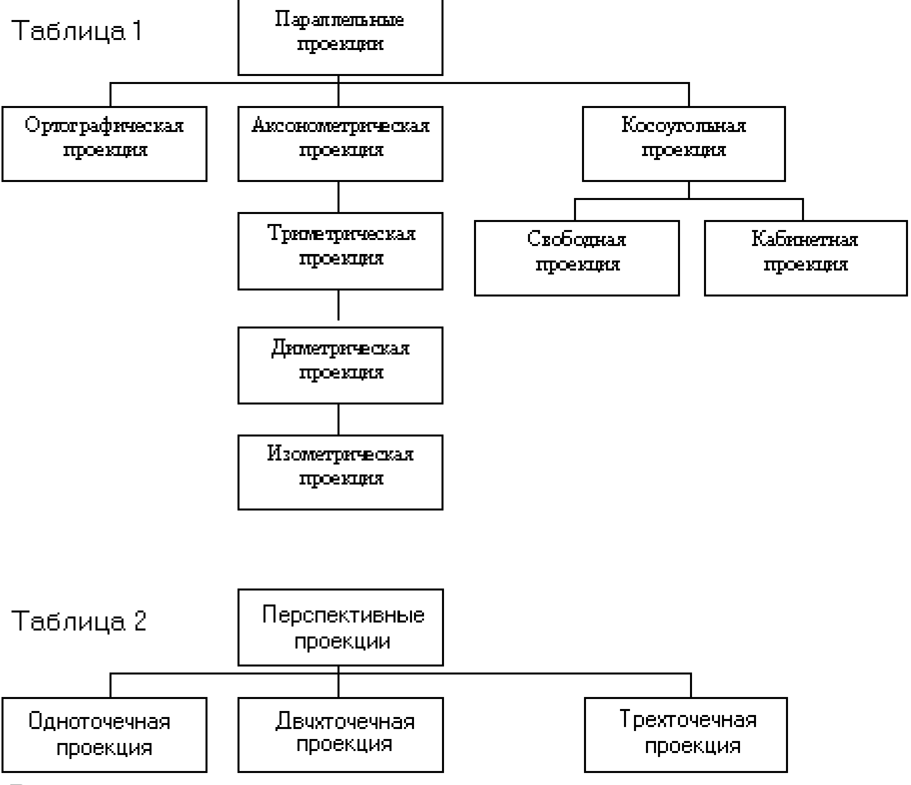


Рис. 24

Каждый из этих двух основных классов разбивается на несколько подклассов в зависимости от взаимного расположения картинной плоскости и координатных осей. Некоторое представление о видах проектирования могут дать приводимые ниже таблицы.



Важное замечание

Использование для описания преобразований проектирования однородных координат и матриц четвертого порядка позволяет упростить изложение и зримо облегчает решение задач геометрического моделирования.

При ортографической проекции картинная плоскость совпадает с одной из координатных плоскостей или параллельна ей (рис. 25). Матрица проектирования вдоль оси X на плоскость YZ имеет вид:

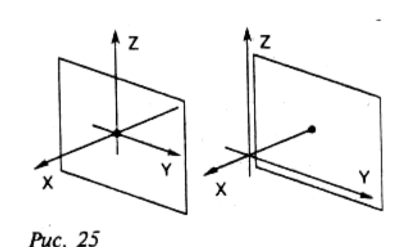


Рис. 25

$$[P_x]=\begin{bmatrix}0&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

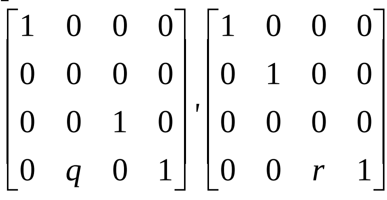
В случае, если плоскость проектирования параллельна координатной плоскости, необходимо умножить матрицу $[P_x]$ на матрицу сдвига. В результате получаем

$$[P_x]\cdot\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\p&0&0&1\end{bmatrix}=\begin{bmatrix}0&0&0&0\\0&1&0&0\\0&0&1&0\\p&0&0&1\end{bmatrix}$$

X
Y
Z

Рис. 26

Аналогично записываются матрицы проектирования вдоль двух других координатных осей:



$$\begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&1&0\\0&q&0&1\end{bmatrix},\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&r&1\end{bmatrix}$$

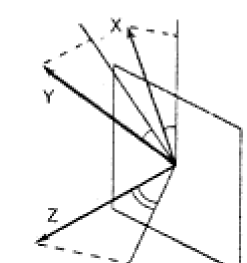


Рис. 27

Замечание

Все три полученные матрицы проектирования вырожденны.

При аксонометрической проекции проектирующие прямые перпендикулярны картинной плоскости.

В соответствии со взаимным расположением плоскости проектирования и координатных осей различают три вида проекций:

- триметрию - нормальный вектор картинной плоскости образует с ортами координатных осей попарно различные углы (рис. 26);
- диметрию - два угла между нормалью картинной плоскости и координатными осями равны (рис. 27);

X
Y
Z

Рис. 28



- изометрию - все три угла между нормалью картинной плоскости и координатными осями равны (рис. 28).

Каждый из трех видов указанных проекций получается комбинацией поворотов, за которой следует параллельное проектирование.

При повороте на угол ψ относительно оси ординат, на угол φ вокруг оси абсцисс и последующего проектирования вдоль оси аппликат возникает матрица

$$[M]=\begin{bmatrix}\cos\psi&\sin\varphi\sin\psi&0&0\\0&\cos\psi&0&0\\\sin\psi&-\sin\psi\cos\psi&0&0\\0&0&0&1\end{bmatrix}==\begin{bmatrix}\cos\psi&0&-\sin\psi&0\\0&1&0&0\\\sin\psi&0&\cos\psi&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&\cos\varphi&\sin\varphi&0\\0&-\sin\varphi&\cos\varphi&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

Покажем, как при этом преобразуются единичные орты координатных осей *X*, *Y*, *Z*:

$$(1\quad 0\quad 0\quad 1)[M]=(\cos\psi\quad \sin\varphi\sin\psi\quad 0\quad 1),$$

$$(0\quad 1\quad 0\quad 1)[M]=(0\quad \cos\varphi\quad 0\quad 1),$$

$$(0\quad 0\quad 1\quad 1)[M]=(\sin\psi\quad -\sin\varphi\cos\psi\quad 0\quad 1).$$

Диметрия характеризуется тем, что длины двух проекций совпадают:

$$\cos^2\psi+\sin^2\varphi\sin^2\psi=\cos^2\varphi.$$

Отсюда следует, что

$\sin^2\psi = \tan^2\varphi$.

В случае изометрии имеем

$\cos^2\psi + \sin^2\varphi \sin^2\psi = \cos^2\varphi,$

$\sin^2\psi + \sin^2\varphi \cos^2\psi = \cos^2\varphi.$

Из последних двух соотношений вытекает, что

$\sin^2\varphi = \frac{1}{3}, \sin^2\psi = \frac{1}{2}$.

При триметрии длины проекций попарно различны.

Проекции, для получения которых используется пучок прямых, не перпендикулярных плоскости экрана, принято называть косоугольными.

При косоугольном проектировании орта оси Z на плоскость XY (рис. 29) имеем:

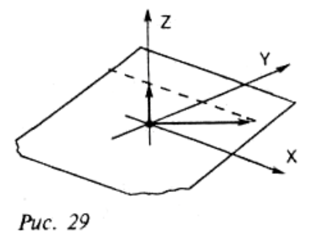


Рис. 29

$(0 \quad 0 \quad 1 \quad 1) \to (\alpha \quad \beta \quad 0 \quad 1)$.

Матрица соответствующего преобразования имеет следующий вид:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \alpha & \beta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Выделяют два вида косоугольных проекций: свободную проекцию (угол наклона проектирующих прямых к плоскости экрана равен половине прямого) и кабинетную проекцию (частный случай свободной проекции - масштаб по третьей оси вдвое меньше).

В случае свободной проекции

$\alpha = \beta = \cos\frac{\pi}{4}$,

в случае кабинетной -

$\alpha = \beta = \frac{1}{2}\cos\frac{\pi}{4}$.

Перспективные (центральные) проекции строятся более сложно

Предположим для простоты, что центр проектирования лежит на оси Z в точке C(0, 0, c) и плоскость проектирования совпадает с координатной плоскостью XY (рис. 30). Возьмем в пространстве произвольную точку M(x, y, z), проведем через нес и точку C прямую и запишем соответствующие параметрические уравнения.

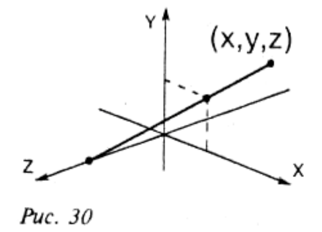


Рис. 30

Имеем:

$X^* = xt, Y^* = yt, Z^* = c + (z - c)t$.

Найдем координаты точки пересечения построенной прямой с плоскостью XY. Из условия $Z^* = 0$ получаем, что

$$t^* = \frac{1}{1 - \frac{z}{c}}$$

и далее

$$X^* = \frac{1}{1 - \frac{z}{c}} x, Y^* = \frac{1}{1 - \frac{z}{c}} y.$$

Интересно заметить, что тот же самый результат можно получить, привлекая матрицу

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

В самом деле, переходя к однородным координатам, прямым вычислением совсем легко проверить, что

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{pmatrix} x & y & 0 & 1-\frac{z}{c} \end{pmatrix}$$

Вспоминая свойства однородных координат, запишем полученный результат в несколько ином виде:

$$\begin{pmatrix} \dfrac{x}{1-\frac{z}{c}} & \dfrac{y}{1-\frac{z}{c}} & 0 & 1 \end{pmatrix}$$

и затем путем непосредственного сравнения убедимся в том, что это координаты той же самой точки.

Замечание

Матрица проектирования, разумеется, вырожденна.

Матрица соответствующего перспективного преобразования (без проектирования) имеет следующий вид:

$$[Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(обратим внимание на то, что последняя матрица невырожденна).

Рассмотрим пучок прямых, параллельных оси Z, и попробуем разобраться в том, что с ним происходит под действием матрицы [Q].

Каждая прямая пучка однозначно определяется точкой (скажем, M(x, y, 0)) своего пересечения с плоскостью XY и описывается уравнениями

X = x, Y = y, Z = t.

Переходя к однородным координатам и используя матрицу [Q], получаем

$$\begin{pmatrix} x & y & t & 1 \end{pmatrix} [Q] = \begin{pmatrix} x & y & t & 1-\frac{t}{c} \end{pmatrix}$$

или, что то же,

$$\begin{pmatrix} \dfrac{x}{1-\frac{t}{c}} & \dfrac{y}{1-\frac{t}{c}} & -c\dfrac{t}{t-c} & 1 \end{pmatrix}$$

Устремим t в бесконечность.

При переходе к пределу точка (x y t 1) преобразуется в (0 0 1 0). Чтобы убедиться в этом, достаточно разделить каждую координату на t:

$\left(\frac{x}{t} \quad \frac{y}{t} \quad 1 \quad \frac{1}{t}\right).$

Точка (0 0 -c 1) является пределом (при t, стремящемся к бесконечности) правой части

$$\left(\frac{x}{1-\frac{t}{c}} \quad \frac{y}{1-\frac{t}{c}} \quad -c\frac{t}{t-c} \quad 1\right)$$

рассматриваемого равенства.

Тем самым, бесконечно удаленный (несобственный) центр $(0 \quad 0 \quad 1 \quad 0)$ пучка прямых, параллельных оси Z, переходит в точку (0 0 -c 1) оси Z.

Вообще, каждый несобственный пучок прямых (совокупность прямых, параллельных заданному направлению), не параллельный картинной плоскости,

X = x + lt, Y = y + mt, Z = nt, $n \neq 0$

под действием преобразования, задаваемого матрицей [Q], переходит в собственный пучок

$$(x+lt \quad y+nt \quad nt \quad 1)[Q] = \left(x+lt \quad y+mt \quad nt \quad 1-\frac{nt}{c}\right)$$

Центр этого пучка

$$\left(-\frac{lc}{n} \quad -\frac{mc}{n} \quad -c \quad 1\right)$$

называют точкой схода.

Принято выделять так называемые главные точки схода, которые соответствуют пучкам прямых, параллельных координатным осям.

Для преобразования с матрицей [Q] существует лишь одна главная точка схода (рис.31). В общем случае (когда оси координатной системы не параллельны плоскости экрана) таких точек три. Матрица соответствующего преобразования выглядит следующим образом:

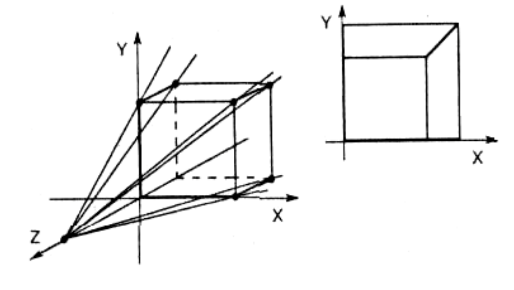


Рис. 31

$$\begin{bmatrix} 1 & 0 & 0 & -1/a \\ 0 & 1 & 0 & -1/b \\ 0 & 0 & 1 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Пучок прямых, параллельных оси

OX *OY*

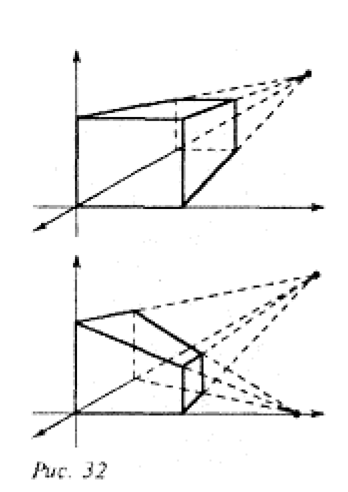
$(1 \quad 0 \quad 0 \quad 0)$ $(0 \quad 1 \quad 0 \quad 0)$

переходит в пучок прямых с центром

$\left(1 \quad 0 \quad 0 \quad -\frac{1}{a}\right)$ $\left(0 \quad 1 \quad 0 \quad -\frac{1}{b}\right)$

или, что то же,

$(-a \quad 0 \quad 0 \quad 1)$ $(-b \quad 0 \quad 0 \quad 1)$


Рис. 32

Точки (-a, 0, 0) и (0, -b, 0) суть главные точки схода.

На рис. 32 изображены проекции куба со сторонами, параллельными координатным осям, с одной и с двумя главными точками схода.

Особенности проекций гладких отображений

В заключение этой главы мы остановимся на некоторых эффектах, возникающих при проектировании искривленных объектов (главным образом поверхностей) на картинную плоскость.

Важно отметить, что описываемые ниже эффекты возникают вне зависимости от того, является ли проектирование параллельным или центральным.

Будем считать для простоты, что проектирование проводится при помощи пучка параллельных прямых, идущих перпендикулярно картинной плоскости, а система координат (X, Y, Z) в пространстве выбрана так, что картинная плоскость совпадает с координатной плоскостью X = 0.

Укажем три принципиально различных случая.

1-й случай

Заданная поверхность - плоскость, описываемая уравнением Z = X и проектируемая на плоскость X = 0 (рис. 33). Записав ее уравнение в неявном виде

X-Z=0,

Вектор L , вдоль которого осуществляется проектирование, имеет координаты

вычислим координаты нормального вектора. Имеем:

$\vec{N} = (1, \quad 0, \quad -1)$.

$\vec{L} = (1, \quad 0, \quad 0)$.

Рис. 33

Легко видеть, что скалярное произведение этих двух векторов отлично от нуля:

$(\vec{N}, \vec{L}) = 1 > 0$.

Тем самым вектор проектирования и нормальный вектор рассматриваемой поверхности не перпендикулярны ни в одной точке. Отметим, что полученная проекция особенностей не имеет.



Рис. 34

2-й случай

Заданная поверхность - параболический цилиндр с уравнением

$\vec{N} = (2X, \quad 0 \quad -1)$

ортогонален вектору проектирования $\vec{L}$ в точках оси Y. Это вытекает из того, что

$(\vec{N}, \vec{L}) = 2X$.

Z = X^2 или, что то же, X^2 - Z = 0. Нормальный вектор

Здесь, в отличие от первого случая, точки плоскости X = 0 разбиваются на три класса:

- к первому относятся точки (Z > 0), у которых два прообраза (на рис. 34 этот класс заштрихован);
- ко второму - те, у которых прообраз один (Z = 0);
- и наконец, к третьему классу относятся точки, у которых прообразов на цилиндре нет вовсе.

Прямая X = O, Z = 0 является особой.

Вдоль нее векторы $\vec{N}$ и $\vec{L}$ ортогональны.

Особенность этого типа называется складкой.

3-й случай

Рассмотрим поверхность, заданную уравнением $Z = X + XY$ или, что то же,

$X^3 + XY - Z = 0$.

Вычислим нормальный вектор этой поверхности

$N = (3X^2 + Y,\ X, -1)$

и построим ее, применив метод сечений.

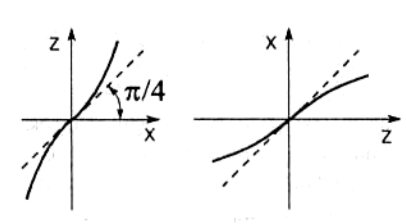


Рис. 35

Пусть $Y = 1$. Тогда

$Z = X^3 + X$

(рис. 35).

При $Y = 0$ имеем:

$Z = Xx^3$

(рис. 36).

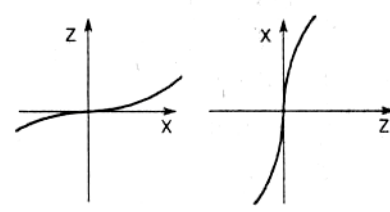


Рис. 36

Наконец,при $Y = -1$ получаем:

$Z = X^3 - X$

(рис. 37).

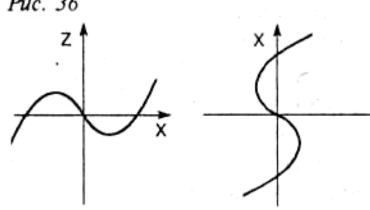


Рис. 37

Построенные сечения дают представление обо всей поверхности. Поэтому нарисовать ее теперь уже несложно (рис. 38).

Из условия

$(\vec{N}, \vec{L}) = 3X^2 + Y = 0$

и уравнения поверхности получаем, что вдоль лежащей на ней кривой с уравнениями

$Y = -3X^2,\ Z = -2X^3$

вектор проектирования $\vec{L}$ и нормальный вектор $\vec{N}$ рассматриваемой поверхности ортогональны. Исключая X, получаем, что

$(-Y/3)^3 = (-Z/2)^2$ или

$27Z^2 = -4Y^3$.

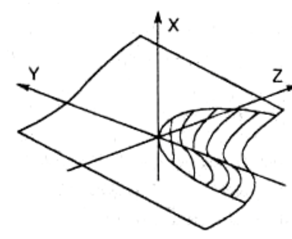


Рис. 38

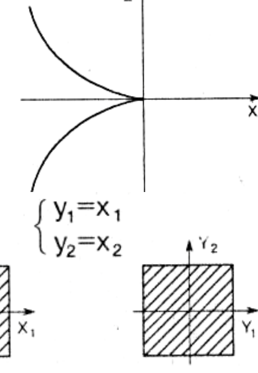
Последнее равенство задает на координатной плоскости $X = 0$ полукубическую параболу (рис. 39), которая делит точки этой плоскости на три класса: к первому относятся точки, лежащие на острие (у каждой из них на заданной поверхности ровно два прообраза), внутри острия лежат точки второго класса (каждая точка имеет по три прообраза), а вне - точки третьего класса, имеющие по одному прообразу. Особенность этого типа называется сборкой.

Замечание

Возникающая в третьем случае полукубическая парабола имеет точку заострения. Однако ее прообраз

$X = X,\ Y = -3X^2,\ Z = -2X^3$

является регулярной кривой, лежащей на заданной поверхности.



В теории особенностей (теории катастроф) доказывается: при проектировании на плоскость произвольного гладкого объекта - поверхности возможны (с точностью до малого шевеления, рассыпающего более сложные проекции) только три указанных типа проекции - обыкновенная проекция, складка и сборка.

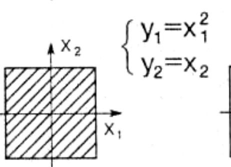
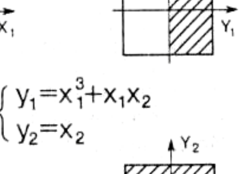



Рис. 40

Сказанное следует понимать так: при проектировании гладких поверхностей на плоскость могут возникать и другие, более сложные особенности. Однако в отличие от трех перечисленных выше все они

оказываются неустойчивыми - при малых изменениях либо направления проектирования, либо взаимного расположения плоскости и проектируемой поверхности эти особенности не сохраняются и переходят в более простые.

Замечание

По существу, в приведенных примерах рассмотрены три типа отображения 2-плоскости в 2-плоскость (рис. 40).

Использование средств языка C++ для работы с векторами и преобразованиями

Язык C++ предоставляет очень удобные средства, позволяющие заметно упростить работу с векторами и преобразованиями в пространстве.

Рассмотрим реализацию работы с векторами.

```
// File Vector.h
#ifndef   __VECTOR__
#define   __VECTOR__

#include  <math.h>

class     Vector
{
public:
   double x, y, z;

   Vector () {};
   Vector ( double v ) { x = y = z = v; };
   Vector ( const Vector& v ) { x = v.x; y = v.y; z = v.z; };
   Vector ( double vx, double vy, double vz ) { x = vx; y = vy; z = vz; };

   Vector& operator = ( const Vector& v ) { x = v.x; y = v.y; z = v.z; return *this; };
   Vector& operator = ( double f ) { x = y = z = f; return *this; };
   Vector  operator - () const;

   Vector& operator += ( const Vector& );
   Vector& operator -= ( const Vector& );
   Vector& operator *= ( const Vector& );
   Vector& operator *= ( double );
   Vector& operator /= ( double );

   friend Vector operator + ( const Vector&, const Vector& );
   friend Vector operator - ( const Vector&, const Vector& );
   friend Vector operator * ( const Vector&, const Vector& );
   friend Vector operator * ( double, const Vector&  );
   friend Vector operator * ( const Vector&, double  );
   friend Vector operator / ( const Vector&, double  );
   friend Vector operator / ( const Vector&, const Vector& );
   friend double operator & ( const Vector& u, const Vector& v ) { return u.x*v.x + u.y*v.y + u.z*v.z;
};
   friend Vector operator ^ ( const Vector&, const Vector& );

   double operator !  () { return (double) sqrt ( x*x + y*y + z*z ); };
   double& operator [] ( int n ) { return * ( &x + n ); };
   int    operator <  ( double v ) { return x < v && y < v && z < v; };
   int    operator >  ( double v ) { return x > v && y > v && z > v; };
};

class     Ray
{
public:
   Vector Org;
   Vector Dir;                        // direction must be normalyzed
```

```
    Ray () {};
    Ray ( Vector& o, Vector& d ) { Org = o; Dir = d; };

    Vector Point ( double t ) { return Org + Dir*t; };
};

/////////////////// implementation ////////////////////////

inline    Vector Vector :: operator - () const
{
    return Vector ( -x, -y, -z );
}

inline    Vector operator + ( const Vector& u, const Vector& v )
{
    return Vector ( u.x + v.x, u.y + v.y, u.z + v.z );
}

inline    Vector operator - ( const Vector& u, const Vector& v )
{
    return Vector ( u.x - v.x, u.y - v.y, u.z - v.z );
}

inline    Vector operator * ( const Vector& u, const Vector& v )
{
    return Vector ( u.x * v.x, u.y * v.y, u.z * v.z );
}

inline    Vector operator * ( const Vector& u, double f )
{
    return Vector ( u.x * f, u.y * f, u.z * f );
}

inline    Vector operator * ( double f, const Vector& v )
{
    return Vector ( f * v.x, f * v.y, f * v.z );
}

inline    Vector operator / ( const Vector& v, double f )
{
    return Vector ( v.x / f, v.y / f, v.z / f );
}

inline    Vector operator / ( const Vector& u, const Vector& v )
{
    return Vector ( u.x / v.x, u.y / v.y, u.z / v.z );
}

inline    Vector& Vector :: operator += ( const Vector& v )
{
    x += v.x;
    y += v.y;
    z += v.z;

    return *this;
}

inline    Vector& Vector :: operator -= ( const Vector& v )
{
    x -= v.x;
    y -= v.y;
    z -= v.z;

    return *this;
}
```

```
inline    Vector&Vector :: operator *= ( double v )
{
   x *= v;
   y *= v;
   z *= v;

   return *this;
}

inline    Vector&Vector :: operator *= ( const Vector& v )
{
   x *= v.x;
   y *= v.y;
   z *= v.z;

   return *this;
}

inline    Vector&Vector :: operator /= ( double v )
{
   x /= v;
   y /= v;
   z /= v;

   return *this;
}

////////////////////////// Functions ////////////////////////////////

inline    Vector Normalize ( Vector& v ) { return v / !v; };
Vector    RndVector ();
Vector&   Clip ( Vector& );
#endif
```

С этой целью создается класс Vector, содержащий в себе компоненты вектора, и для этого класса переопределяются основные знаки операций:

- \- унарный минус и поэлементное вычитание векторов;

\+ - поэлементное сложение векторов;

* - умножение вектора на число;

* - поэлементное умножение векторов;

/ - деление вектора на число;

/ - поэлементное деление векторов;

& - скалярное произведение векторов;

^ - векторное произведение;

! - длина вектора;

[] - компонента вектора.

При этом стандартные приоритеты операций сохраняются. Кроме этих операций определяются также некоторые простейшие функции для работы с векторами:

- Normalize - нормирование вектора;
- RndVector - получение почти равномерно распределенного случайного единичного вектора;
- Clip- отсечение вектора.

С использованием этого класса можно в естественной и удобной форме записывать сложные векторные выражения.

Аналогичным образом вводится класс Matrix, служащий для представления матриц преобразований в трехмерном пространстве. Для этого класса также производится переопределение основных знаков операций.

```
// File Matrix.h
#ifndef __MATRIX__
#define __MATRIX__

#include "Vector.h"

class Matrix
{
public:
   double x [4][4];

   Matrix () {};
   Matrix ( double );

   Matrix& operator += ( const Matrix& );
   Matrix& operator -= ( const Matrix& );
   Matrix& operator *= ( const Matrix& );
   Matrix& operator *= ( double );
   Matrix& operator /= ( double );

   void  Invert ();
   void  Transpose ();
   friendMatrix operator + ( const Matrix&, const Matrix& );
   friendMatrix operator - ( const Matrix&, const Matrix& );
   friendMatrix operator * ( const Matrix&, double );
   friendMatrix operator * ( const Matrix&, const Matrix& );
   friendVector operator * ( const Matrix&, const Vector& );
};

Matrix   Translate ( const Vector& );
Matrix   Scale     ( const Vector& );
Matrix   RotateX   ( double );
Matrix   RotateY   ( double );
Matrix   RotateZ   ( double );
Matrix   Rotate    ( const Vector& v, double );
Matrix   MirrorX   ();
Matrix   MirrorY   ();
Matrix   MirrorZ   ();

#endif
```

```
// File Matrix.cpp
#include <math.h>
#include "Matrix.h"

Matrix :: Matrix ( double v )
{
   for ( int i = 0; i < 4; i++)
         for ( int j = 0; j < 4; j++)
               x [i][j] = (i == j) ? v : 0.0;

   x [3][3] = 1;
}

void     Matrix :: Invert ()
{
   MatrixOut ( 1 );

   for ( int i = 0; i < 4; i++ )
   {
         doubled = x [i][i];
```

```
        if ( d != 1.0)
        {
            for ( int j = 0; j < 4; j++ )
            {
                Out.x [i][j] /= d;
                x [i][j]     /= d;
            }
        }

        for ( int j = 0; j < 4; j++ )
        {
            if ( j != i )
            {
                if ( x [j][i] != 0.0)
                {
                    doublemulby = x[j][i];

                    for ( int k = 0; k < 4; k++ )
                    {
                        x [j][k]     -= mulby * x [i][k];
                        Out.x [j][k] -= mulby * Out.x [i][k];
                    }
                }
            }
        }
    }

    *this = Out;
}

void     Matrix :: Transpose ()
{
    doublet;

    for ( int i = 0; i < 4; i++ )
        for ( int j = i; j < 4; j++ )
            if ( i != j )
            {
                t       = x [i][j];
                x [i][j] = x [j][i];
                x [j][i] = t;
            }
}

Matrix&  Matrix :: operator += ( const Matrix& A )
{
    for ( int i = 0; i < 4; i++ )
        for ( int j = 0; j < 4; j++ )
            x [i][j] += A.x [i][j];

    return *this;
}

Matrix&  Matrix :: operator -= ( const Matrix& A )
{
    for ( int i = 0; i < 4; i++ )
        for ( int j = 0; j < 4; j++ )
            x [i][j] -= A.x [i][j];

    return *this;
}

Matrix&  Matrix :: operator *= ( double v )
{
    for ( int i = 0; i < 4; i++ )
```

```
            for ( int j = 0; j < 4; j++ )
                  x [i][j] *= v;

      return *this;
   }

   Matrix&  Matrix :: operator *= ( const Matrix& A )
   {
      Matrixres = *this;

      for ( int i = 0; i < 4; i++ )
            for ( int j = 0; j < 4; j++ )
            {
                  double sum = 0;

                  for ( int k = 0; k < 4; k++ )
                        sum += res.x [i][k] * A.x [k][j];

                  x [i][j] = sum;
            }

       return *this;
   }

   Matrix   operator + ( const Matrix& A, const Matrix& B )
   {
      Matrixres;

      for ( int i = 0; i < 4; i++ )
            for ( int j = 0; j < 4; j++ )
                  res.x [i][j] = A.x [i][j] + B.x [i][j];

      return res;
   }

   Matrix   operator - ( const Matrix& A, const Matrix& B )
   {
      Matrixres;

      for ( int i = 0; i < 4; i++ )
            for ( int j = 0; j < 4; j++ )
                  res.x [i][j] = A.x [i][j] - B.x [i][j];

      return res;
   }

   Matrix   operator * ( const Matrix& A, const Matrix& B )
   {
      Matrixres;

      for ( int i = 0; i < 4; i++ )
            for ( int j = 0; j < 4; j++ )
            {
                  double sum = 0;

                  for ( int k = 0; k < 4; k++ )
                        sum += A.x [i][k] * B.x [k][j];

                  res.x [i][j] = sum;
            }

      return res;
   }

   Matrix   operator * ( const Matrix& A, double v )
   {
```

```
    Matrixres;

    for ( int i = 0; i < 4; i++ )
        for ( int j = 0; j < 4; j++ )
            res.x [i][j] = A.x [i][j] * v;

    return res;
}

Vector  operator * ( const Matrix& M, const Vector& v )
{
    Vectorres;

    res.x = v.x * M.x [0][0] + v.y * M.x [1][0] + v.z * M.x [2][0] + M.x [3][0];
    res.y = v.x * M.x [0][1] + v.y * M.x [1][1] + v.z * M.x [2][1] + M.x [3][1];
    res.z = v.x * M.x [0][2] + v.y * M.x [1][2] + v.z * M.x [2][2] + M.x [3][2];

    doubledenom = v.x * M.x [0][3] + v.y * M.x [1][3] + v.z * M.x [2][3] + M.x [3][3];

    if ( denom != 1.0 )
        res /= denom;

    return res;
}

//////////////////////// Derived classes ////////////////////////////

Matrix  Translate ( const Vector& Loc )
{
    Matrixres ( 1 );

    res.x [3][0] = Loc.x;
    res.x [3][1] = Loc.y;
    res.x [3][2] = Loc.z;

    return res;
}

Matrix  Scale ( const Vector& v )
{
    Matrixres ( 1 );

    res.x [0][0] = v.x;
    res.x [1][1] = v.y;
    res.x [2][2] = v.z;

    return res;
}

Matrix  RotateX ( double Angle )
{
    Matrixres ( 1 );
    double  Cosine = cos ( Angle );
    double  Sine   = sin ( Angle );

    res.x [1][1] = Cosine;
    res.x [2][1] = -Sine;
    res.x [1][2] = Sine;
    res.x [2][2] = Cosine;

    return res;
}

Matrix  RotateY ( double Angle )
{
    Matrixres ( 1 );
```

```
    double  Cosine = cos ( Angle );
    double  Sine   = sin ( Angle );

    res.x [0][0] = Cosine;
    res.x [2][0] = -Sine;
    res.x [0][2] = Sine;
    res.x [2][2] = Cosine;

    return res;
}

Matrix   RotateZ ( double Angle )
{
    Matrixres ( 1 );
    double  Cosine = cos ( Angle );
    double  Sine   = sin ( Angle );

    res.x [0][0] = Cosine;
    res.x [1][0] = -Sine;
    res.x [0][1] = Sine;
    res.x [1][1] = Cosine;

    return res;
}

Matrix   Rotation ( const Vector& axis, double angle )
{
    Matrixres ( 1 );
    double  Cosine = cos ( angle );
    double  Sine   = sin ( angle );

    res.x [0][0] = axis.x * axis.x + ( 1 - axis.x * axis.x ) * Cosine;
    res.x [0][1] = axis.x * axis.y * ( 1 - Cosine ) + axis.z * Sine;
    res.x [0][2] = axis.x * axis.z * ( 1 - Cosine ) - axis.y * Sine;
    res.x [0][3] = 0;

    res.x [1][0] = axis.x * axis.y * ( 1 - Cosine ) - axis.z * Sine;
    res.x [1][1] = axis.y * axis.y + ( 1 - axis.y * axis.y ) * Cosine;
    res.x [1][2] = axis.y * axis.z * ( 1 - Cosine ) + axis.x * Sine;
    res.x [1][3] = 0;

    res.x [2][0] = axis.x * axis.z * ( 1 - Cosine ) + axis.y * Sine;
    res.x [2][1] = axis.y * axis.z * ( 1 - Cosine ) - axis.x * Sine;
    res.x [2][2] = axis.z * axis.z + ( 1 - axis.z * axis.z ) * Cosine;
    res.x [2][3] = 0;

    res.x [3][0] = 0;
    res.x [3][1] = 0;
    res.x [3][2] = 0;
    res.x [3][3] = 1;

    return res;
}

Matrix   MirrorX ()
{
    Matrixres ( 1 );

    res.x [0][0] = -1;

    return res;
}

Matrix   MirrorY ()
{
    Matrixres ( 1 );
```

```
    res.x [1][1] = -1;

    return res;
}

Matrix   MirrorZ ()
{
    Matrixres ( 1 );

    res.x [2][2] = -1;

    return res;
}
```

РАСТРОВЫЕ АЛГОРИТМЫ

Так как в подавляющем большинстве графические устройства являются растровыми, то есть представляют изображение в виде прямоугольной матрицы (сетки, целочисленной решетки) пикселов (растра), то, естественно, возникает необходимость в растровых алгоритмах.

Хотя большинство графических библиотек содержат внутри себя достаточное количество простейших растровых алгоритмов, таких, как:

- переведение идеального объекта (отрезка, окружности и др.) в их растровые образы;
- обработка растровых изображений,

тем не менее часто возникает необходимость явного построения растровых алгоритмов.

Достаточно важным понятием для растровой сетки является связность - возможность соединения двух пикселов растровой линией, то есть последовательным набором пикселов. При этом возникает вопрос, когда пикселы (x_1, y_1) и (x_2, y_2) можно считать соседними.

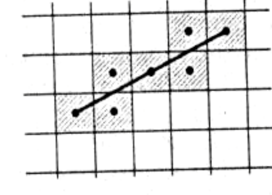
Вводится два понятия связности:

$|x_1 - x_2| + |y_1 - y_2| \le 1$,

8-связность, когда пикселы считаются соседними, если их x- и y-координаты отличаются не более чем на единицу, то есть

4-связность, когда пикселы считаются соседними, если либо их x-координаты, либо y-координаты отличаются на единицу, то есть

$|x_1 - x_2| \le 1, |y_1 - y_2| \le 1$.


Рис. 1

Понятие 4-связности является более сильным, чем 8-связность: любые два 4-связных пиксела являются и 8-связными, но не наоборот (рис. 1).

В качестве линии на растровой сетке выступает набор пикселов P_1, P_2,..., P_n, где любые два пиксела P_i; P_{i+1} являются соседними.

Замечание

Так как понятие линии базируется на понятии связности, то естественным образом возникает понятие 4- и 8-связных линий. Поэтому когда мы говорим о растровом представлении, например, отрезка, то следует ясно понимать, о каком именно представлении идет речь. При этом нужно иметь в виду, что растровое представление объекта не является единственным и возможны различные способы построения.

Растровое представление отрезка. Алгоритм Брезенхейма

Рассмотрим задачу построения растрового изображения отрезка, соединяющего точки (x_1, y_1) и (x_2, y_2). Для простоты будем считать, что $0 =< y_2 - y_1 =< x_2 - x_1$. Тогда отрезок описывается следующим уравнением:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1), x \in [x_1, x_2]$$

или

$y = kx + b$.

Простейший алгоритм растрового представления отрезка имеет вид:

```
// File Line1.cpp
void Line ( int x1, int y1, int x2, int y2, int color )
{
double k = ((double)(y2-yl))/(x2-xl):
double b = y1 - k*x1;
for ( int x = x1; x <= x2; x++ )
putpixel ( x, round ( k*x + b ), color );
}
```

Используя рекуррентное соотношение для вычисления y, можно упростить функцию, однако это не устраняет основного недостатка алгоритма - использования вещественных вычислений для работы на целочисленной решетке.

В 1965 году Брезенхеймом был предложен простой целочисленный алгоритм для растрового построения отрезка, первоначально предназначенный для использования в графопостроителях.

При построении растрового изображения отрезка всегда выбирается ближайший по вертикали пиксел.

При этом из двух точек A и B (рис. 2) выбирается та, которая ближе к исходной прямой (в данном случае выбирается точка A, так как a < b). Для этого вводится число d, равное $(x_2 - x_1)(b - a)$.

В случае d > 0 значение y от предыдущей точки увеличивается на 1, а d -на $2(\Delta y - \Delta x)$. В противном случае значение y не изменяется, а значение d заменяется на $2\Delta y$.

Реализация приведенного алгоритма представлена ниже.

```
// File Line2.cpp
// simplest Bresenhames alg. 0 <= y2 - y1 <= x2 - x1
void     Line2 ( int x1, int y1, int x2, int y2, int color )
{
   int   dx = x2 - x1;
   int   dy = y2 - y1;
   int   d  = ( dy << 1 ) - dx;
   int   d1 = dy << 1;
   int   d2 = ( dy - dx ) << 1;

   putpixel ( x1, y1, color );

   for ( int x = x1 + 1, y = y1; x <= x2; x++ )
   {
         if ( d > 0 )
         {
               d += d2;
               y += 1;
         }
         else
               d += d1;

         putpixel ( x, y, color );
   }
}
```

Из предложенного примера несложно написать функцию для построения 4-связной развертки отрезка.

```
// File Line3.cpp
void     Line4 ( int x1, int y1, int x2, int y2, int color )
{
   int   dx = x2 - x1;
   int   dy = y2 - y1;
   int   d  = 0;
   int   d1 = dy << 1;
   int   d2 = -( dx << 1 );

   putpixel ( x1, y1, color );

   for ( int x = x1 , y = y1; i = 1; i <= dx + dy; i++ )
   {
         if ( d > 0 )
         {
               d += d2;
               y += 1;
         }
```

```
            else
            {
                  d += d1;
                  x += 1;
            }
            putpixel ( x, y, color );
      }
}
```

Общий случай произвольного отрезка легко сводится к рассмотренному выше, следует только иметь в виду, что при выполнении неравенства $|\Delta y| \geq |\Delta x|$ необходимо поменять местами x и y.

```
// File Line4.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

// Bresenhames alg.
void     Line ( int x1, int y1, int x2, int y2, int color )
{
   int   dx = abs ( x2 - x1 );
   int   dy = abs ( y2 - y1 );
   int   sx = x2 >= x1 ? 1 : -1;
   int   sy = y2 >= y1 ? 1 : -1;

   if ( dy <= dx )
   {
         int   d  = ( dy << 1 ) - dx;
         int   d1 = dy << 1;
         int   d2 = ( dy - dx ) << 1;

         putpixel ( x1, y1, color );

         for ( int x = x1 + sx, y = y1, i = 1; i <= dx; i++, x += sx )
         {
               if ( d > 0 )
               {
                     d += d2;
                     y += sy;
               }
               else
                     d += d1;

               putpixel ( x, y, color );
         }
   }
   else
   {
         int   d  = ( dx << 1 ) - dy;
         int   d1 = dx << 1;
         int   d2 = ( dx - dy ) << 1;

         putpixel ( x1, y1, color );

         for ( int x = x1, y = y1 + sy, i = 1; i <= dy; i++, y += sy )
         {
               if ( d > 0 )
               {
                     d += d2;
                     x += sx;
               }
               else
                     d += d1;
```

```
                putpixel ( x, y, color );
            }
        }
    }

    main ()
    {
        int   driver = DETECT;
        int   mode;
        int   res;

        initgraph ( &driver, &mode, "" );
        if ( ( res = graphresult () ) != grOk )
        {
            printf("\nGraphics error: %s\n", grapherrormsg ( res) );
            exit ( 1 );
        }

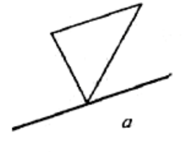
        int   x1 = 501,
              y1 = 100 ,
              x2 = 150,
              y2 = 301;

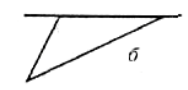
        Line ( x1, y1, x2, y2, WHITE );

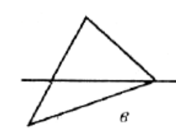
        getch ();
        closegraph ();
    }
```

Отсечение отрезка. Алгоритм Сазерленда-Кохена

Необходимость отсечь выводимое изображение по границам некоторой области встречается довольно часто. В простейших ситуациях в качестве такой области, как правило, выступает прямоугольник.







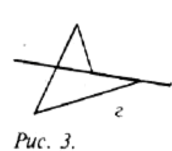


Рис. 3.

Ниже рассматривается достаточно простой и эффективный алгоритм отсечения отрезков по границе произвольного прямоугольника. Он заключается в разбиении всей плоскости на 9 областей прямыми, образующими прямоугольник. В каждой из этих областей все точки по отношению к прямоугольнику расположены одинаково. Определив, в какие области попали концы рассматриваемого отрезка, легко понять, где именно необходимо отсечение. Для этого каждой области сообщается 4-битовый код, где установленный бит 0 означает, что точка лежит левее прямоугольника, бит 1 означает, что точка лежит выше прямоугольника, бит 2 означает, что точка лежит правее прямоугольника, бит 3 означает, что точка лежит ниже прямоугольника.

Приведенная ниже программа реализует алгоритм Сазерленда-Кохена отсечения отрезка по прямоугольной области.

```
// File clip.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

void     Swap ( int& a, int& b )
{
    int   c;

    c = a;
    a = b;
    b = c;
```

```
}

int      OutCode ( int x, int y, int X1, int Y1, int X2, int Y2 )
{
   int code = 0;

   if ( x < X1 )
         code |= 0x01;

   if ( y < Y1 )
         code |= 0x02;

   if ( x > X2 )
         code |= 0x04;

   if ( y > Y2 )
         code |= 0x08;

   return code;
}

void     ClipLine ( int x1, int y1, int x2, int y2,
            int X1, int Y1, int X2, int Y2 )
{
   int   code1   = OutCode ( x1, y1, X1, Y1, X2, Y2 );
   int   code2   = OutCode ( x2, y2, X1, Y1, X2, Y2 );
   int   inside  = ( code1 | code2 ) == 0;
   int   outside = ( code1 & code2 ) != 0;

   while ( !outside && !inside )
   {
         if ( code1 == 0 )
         {
               Swap ( x1, x2 );
               Swap ( y1, y2 );
               Swap ( code1, code2 );
         }

         if ( code1 & 0x01 )           // clip left
         {
               y1 += (long)(y2-y1)*(X1-x1)/(x2-x1);
               x1  = X1;
         }
         else
         if ( code1 & 0x02 )           // clip above
         {
               x1 += (long)(x2-x1)*(Y1-y1)/(y2-y1);
               y1  = Y1;
         }
         else
         if ( code1 & 0x04 )           // clip right
         {
               y1 += (long)(y2-y1)*(X2-x1)/(x2-x1);
               x1  = X2;
         }
         else
         if ( code1 & 0x08 )           // clip below
         {
               x1 += (long)(x2-x1)*(Y2-y1)/(y2-y1);
               y1  = Y2;
         }

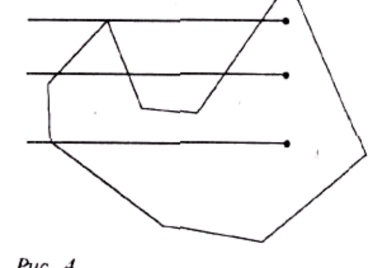
         code1   = OutCode ( x1, y1, X1, Y1, X2, Y2 );
         code2   = OutCode ( x2, y2, X1, Y1, X2, Y2 );
         inside  = ( code1 | code2 ) == 0;
         outside = ( code1 & code2 ) != 0;
```

```
    }

    line ( x1, y1, x2, y2 );
}
```

Определение принадлежности точки многоугольнику

Пусть задан многоугольник, ограниченный несамопересекающейся замкнутой ломаной $P_1P_2...P_n$, и некоторая точка A(x, y) и требуется определить, содержится ли эта точка внутри многоугольника или нет (рис. 4).


Рис. 4

Выпустим из точки A(x, y) произвольный луч и найдем количество точек пересечения этого луча с границей многоугольника. Если отбросить случай, когда луч проходит через какую-либо вершину многоугольника, то решение задачи тривиально - точка лежит внутри, если количество точек пересечения нечетно, и снаружи, если четно.

Ясно, что для любого многоугольника можно построить луч, не проходящий ни через одну из вершин. Однако построение такого луча связано с некоторыми трудностями и, кроме того, проверка пересечения с произвольным лучом сложнее, чем с фиксированным, например горизонтальным.

Возьмем луч, выходящий из точки A, и рассмотрим к чему может привести прохождение луча через вершину. Основные возможные случаи изображены на рис. 3. В простейших случаях а и в, когда ребра, выходящие из соответствующей вершины, лежат либо по разные стороны от луча, либо по одну сторону от луча, четность количества пересечений меняется в первом случае и изменяется во втором. К случаям б и г такой подход непосредственно не годится. Несколько изменим его, заметив, что в случаях а и б вершины, лежащие на луче являются экстремальными значениями в тройке вершин соответствующих отрезков. В других же случаях экстремума нет.

В результате приходим к следующему алгоритму.

Все отрезки, кроме горизонтальных, проверяются на пересечение с горизонтальным лучом, выходящим из точки A. При попадании луча в вершину пересечение засчитывается только с теми отрезками, выходящими из вершины, для которых она является верхней.

```
// File poly.cpp
int PtInPoly ( Point& a, Point * p, int n )
{
int   Count = 0;
for ( int i = 0; i < n; i++ )
{
int j = ( i + 1 ) % n;
if ( p [ i ].y == p [ j ].y ) continue;
if ( P [ i ].y > a.y && p [j].y > a.y ) continue;
if ( P [i].y < a.y && p [j].y < a.y ) continue;
if ( max ( p [i].y, p [j].y ) == a.y ) Count++;
else
if ( min ( p [i].y, p [ j ].y ) == a.y ) continue;
else
{
double t = ( y - p [i].y ) / ( p [j].y - P [i].y );
if (t>0 && t<1 && p[i].x+t-(p[j].x-p[i]x) >= x) Count++;
}
}
return Count & 1;
}
```

Закраска области, заданной цветом границы

Рассмотрим область, ограниченную набором пикселов заданного цвета, и точку (x, y), лежащую внутри этой области.

Задача заполнения области заданным цветом в случае, когда область не является выпуклой, может оказаться довольно сложной.

Простейший алгоритм

```
// File fill1.cpp
void PixelFill ( int x, int y, int BorderColor, int color )
{
int c = getpixel ( x, y );
if ( ( c != BorderColor ) && ( c != color ) )
{
putpixel ( x, y, color );
PixelFill ( x - 1, y, BorderColor, color ); PixelFill ( x + 1, y, BorderColor, color );
PixelFill ( x, y - 1, BorderColor, color ); PixelFill ( x, y + 1, BorderColor, color );
}
```

хотя и абсолютно корректно заполняющий даже самые сложные области, является слишком неэффективным, так как уже для отрисованного пиксела функция вызывается еще три раза, и, кроме того, требует слишком большого стека из-за большой глубины рекурсии.

Поэтому для решения задачи закраски области предпочтительнее алгоритмы, способные обрабатывать сразу целые группы пикселов.

Рассмотрим версию одного из самых популярных алгоритмов подобного типа, когда для заданной точки (x, y) определяется и заполняется максимальный отрезок $[x_l, x_r]$, содержащий эту точку и лежащий внутри области. После этого в поисках еще не заполненных пикселов проверяются отрезки, лежащие выше и ниже. Если такие пикселы находятся, то функция рекурсивно вызывается для их обработки. Этот алгоритм эффективно работает даже для областей с дырками (рис. 5).

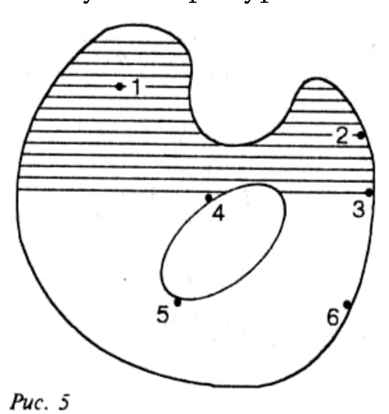


Рис. 5

```
// File fill2.cpp
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

int      BorderColor = WHITE;
int      Color       = GREEN;

int      LineFill ( int x, int y, int dir, int PrevXl, int PrevXr )
{
   int   xl = x;
   int   xr = x;
   int   c;

                        // find line segment
   do
         c = getpixel ( --xl, y );
   while ( ( c != BorderColor ) && ( c != Color ) );

   do
         c = getpixel ( ++xr, y );
   while ( ( c != BorderColor ) && ( c != Color ) );

   xl++;
   xr--;
```

```
    line ( xl, y, xr, y );   // fill segment
                     // fill adjacent segments in the same direction
    for ( x = xl; x<= xr; x++ )
    {
          c = getpixel ( x, y + dir );
          if ( ( c != BorderColor ) && ( c != Color ) )
                x = LineFill ( x, y + dir, dir, xl, xr );
    }

    for ( x = xl; x < PrevXl; x++ )
    {
          c = getpixel ( x, y - dir );
          if ( ( c != BorderColor ) && ( c != Color ) )
                x = LineFill ( x, y - dir, -dir, xl, xr );
    }

    for ( x = PrevXr; x < xr; x++ )
    {
          c = getpixel ( x, y - dir );
          if ( ( c != BorderColor ) && ( c != Color ) )
                x = LineFill ( x, y - dir, -dir, xl, xr );
    }

    return xr;
}

void     Fill ( int x, int y )
{
    LineFill ( x, y, 1, x, x );
}

main ()
{
    int   driver = DETECT;
    int   mode;
    int   res;

    initgraph ( &driver, &mode, "" );
    if ( ( res = graphresult () ) != grOk )
    {
          printf("\nGraphics error: %s\n", grapherrormsg ( res) );
          exit ( 1 );
    }

    circle ( 320, 200, 140 );
    circle ( 260, 200, 40 );
    circle ( 380, 200, 40 );

    getch ();

    setcolor ( Color );

    Fill ( 320, 300 );

    getch ();
    closegraph ();
}
```

Алгоритмы определения точек пересечения произвольного луча с простейшими геометрическими объектами

Эффективные алгоритмы отыскания точки пересечения произвольного луча (ближайшей к его началу) с простейшими двумерными геометрическими объектами (примитивами), такими, как сферы, плоскости, призмы, пирамиды, цилиндры, конусы и

другие, часто оказываются очень полезными при рассмотрении самых разных задач компьютерной графики.

Одной из таких задач-пользователей является метод трассировки лучей, в котором определение точек пересечения лучей с объектами сцены занимает до 95% вычислений, причем значительная доля расчетов падает на лучи, не пересекающие ни один из объектов. Поэтому при обработке сцены этим методом весьма желательно исключить из рассмотрения как можно раньше и как можно больше лучей, избегающих встречи с объектами сцены.

Для этого поступают следующим образом: при помощи попарно непересекающихся простых поверхностей (сфер, плоскостей, выпуклых многогранников и т. п.) локализуют (отделяют друг от друга) сложные объекты сиены и после относительно простых вычислений находят, а затем отбрасывают те лучи, которые не имеют с этими поверхностями общих точек. Ясно, что среди отброшенных не окажется ни одного луча, который бы пересекал объекты исходной сцены. Что же касается оставшихся лучей, то, как правило, они требуют более деликатного обращения.

В этом разделе мы опишем некоторые полезные алгоритмы поиска точек пересечения произвольного луча со сферой, плоскостью, выпуклым многоугольником и прямоугольным параллелепипедом.

Луч с началом в точке О, определяемой начальным вектором

$R_0 = (x_0, y_0, z_0)$

и направляющим вектором $L = (l, m, n) \neq 0$ описывается при помощи параметрического уравнения в векторной форме

$R(t) = R_0 + Lt, t > 0$, или координатными параметрическими уравнениями (рис. 6).

$$\begin{cases} x = x_0 + lt, \\ y = y_0 + mt, (t > 0), \\ z = z_0 + nt \end{cases} \qquad (*)$$

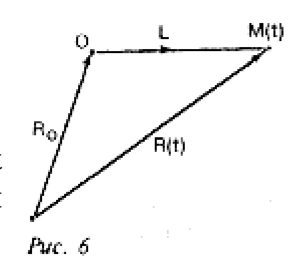


Рис. 6

В случае, если направляющий вектор L заданного луча единичный -

$l^2 + m^2 + n^2 = 1$,

параметр t имеет простой геометрический смысл: его значение t равно расстоянию от начальной точки О заданного луча до его текущей точки M(t), отвечающей этому значению параметра.

1.Пересечение луча со сферой

А. Алгебраическое решение

Сфера радиуса r с центром в точке

$C(x_c, y_c, z_c)$

в прямоугольной декартовой системе координат описывается неявным уравнением вида

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$$

Для того, чтобы найти точку пересечения заданного луча с этой сферой, заменим величины x, y и z в последней формуле их выражениями (*).

В результате несложных преобразований получим уравнение, имеющее следующий вид

$$at^2 + 2bt + c = 0,$$

где

$$a = x_c^2 + y_c^2 + z_c^2;$$

$$b = l(x_0 - x_c) + m(y_0 - y_c) + n(z_0 - z_c);$$

$$c = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2.$$

Замечание

Коэффициент а при квадрате переменной t всегда положителен, и в случае, если направляющий вектор L луча нормирован, равен единице.

Корни полученного квадратного уравнения легко вычисляются:

$$t_- = -b - \sqrt{b^2 - c},$$

$$t_+ = -b + \sqrt{b^2 - c}$$

(при а = 1).

Если дискриминант

$b^2 - c$

отрицателен, то заданный луч проходит мимо сферы. Если же

$b^2 - c >= 0$,

то заданный луч имеет со сферой общую точку, но лишь в том случае,если хотя бы один из корней

t_- или t_+

положителен (напомним условие: $t > 0$).

Наименьший положительный корень (подчеркнем, что $t_- \le t_+$)

определяет на луче ближайшую (считая от начальной точки луча)

точку пересечения луча со сферой.

$M^*(x^*, y^*, z^*)$

пересечения заданного луча со сферой определяются по формулам

$x^* = x_0 + lt^*$,

$y^* = y_0 + mt^*$,

$z^* = z_0 + nt^*$,

а единичный вектор нормали к сфере в этой точке равен

$$N = \frac{1}{r}(x^* - x_C, y^* - y_C, z^* - z_C)$$

Пусть t^* - именно такой корень. Тогда координаты точки

(рис. 7).

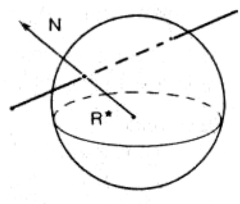


Рис. 7

Перечислим последовательные шаги описанного алгоритма:

шаг 1-й - вычисление коэффициентов a, b и c квадратного уравнения,

шаг 2-й - вычисление дискриминанта и сравнение,



шаг 3-й - вычисление меньшего корня и сравнение,

шаг 4-й - (возможное) вычисление большего корня и сравнение,

шаг 5-й - вычисление точки пересечения,

шаг 6-й - вычисление единичного вектора нормали сферы в точке пересечения -

и укажем характер и количество операций на каждом шаге (в предположении, что некоторые из величин, например

$r^2, \frac{1}{r}$

и т. п. вычислены предварительно):

шаг 1-й - 8 сложений или вычитаний и 7 умножений,

шаг 2-й - 1 вычитание, 2 умножения и 1 сравнение,

шаг 3-й - 1 вычитание, 1 извлечение квадратного корня и 1 сравнение,

шаг 4-й - 1 вычитание и 1 сравнение,

шаг 5-й - 3 сложения и 3 умножения,

шаг 6-й - 3 вычитания и 3 умножения.

Таким образом, общий объем вычислений при отыскании точки пересечения заданного луча с заданной сферой и единичного вектора нормали сферы в этой точке равен

самое большее 17 сложениям или вычитаниям, 15 умножениям, 1 извлечению квадратного корня и 3 сравнениям.

Б. Геометрическое решение

Существует значительное число тестов, показывающих, пересекает ли рассматриваемый луч- заданную сферу или не пересекает. Цель подобных тестов совершенно ясна - избегать громоздких вычислений до тех пор, пока без них уже нельзя будет обойтись.

Для того, чтобы определить, лежит ли начальная точка вне или внутри сферы, достаточно вычислить скалярный квадрат длины вектора, идущего из начальной точки O луча в центр C сферы.

$(OC \cdot OC) < r^2$,

то начальная точка заданного луча лежит внутри сферы.

Если же

$(OC \cdot OC) \geq r^2$,

Если это число меньше квадрата радиуса сферы

то начальная точка луча лежит или вне сферы, или на сфере (рис. 8).

В любом из этих двух случаев следующий шаг состоит в том, чтобы

найти расстояние от начальной точки луча до точки на луче, ближайшей к центру сферы. Нетрудно заметить, что такой точкой является точка пересечения заданного луча с перпендикулярной к нему плоскостью, проходящей через центр сферы.

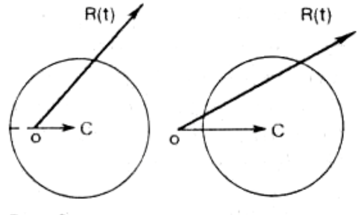


Рис. 8

В результате вычислений получаем точку на луче, отвечающую значению параметра

$t^0 = (OC-L)$. Если

$t^0 < 0$,

то центр сферы лежит как бы позади начальной точки луча.

Для лучей, исходящих из точек, лежащих внутри сферы, это не столь существенно, так как каждый такой луч непременно пересечет сферу (рис. 9, а). Что же касается лучей,

исходящих из точек, лежащих вне сферы, то при выполнении последнего неравенства ни один из них не пересечет заданной сферы, значит, в этом случае тестирование можно считать завершенным (рис. 9, б).

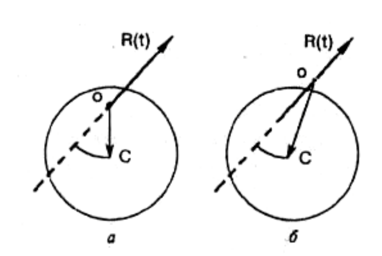


Рис. 9

Пользуясь теоремой Пифагора, вычислим квадрат расстояния о: ближайшей (считая от центра сферы) точки на луче до центра сферы:

$d^2 = (OC \cdot OC) - (OC \cdot L)^2$,

а затем и разность

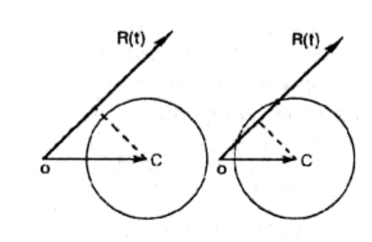
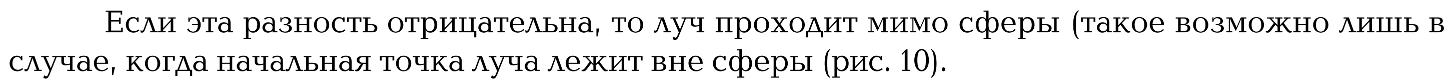


Рис. 10

$r^2 - d^2$.



Если эта разность отрицательна, то луч проходит мимо сферы (такое возможно лишь в случае, когда начальная точка луча лежит вне сферы (рис. 10).

В случае, если разность

$r^2 - a^2$

положительна, остается только найти значение параметра t, соответствующее искомой точке пересечения заданного луча и сферы. Имеем:

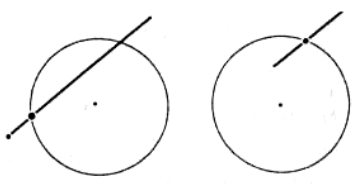


Рис. 11

$t^* = t^0 - \sqrt{r^2 - d^2}$

(для луча с начальной точкой, лежащей вне сферы),

$$t^* = t^0 + \sqrt{r^2 - d^2}$$

(для луча с начальной точкой, лежащей внутри сферы) (рис. 6).

Различие в последних двух формулах объясняется просто: в каждом из этих случаев требуется вычислить разные точки.

Луч, начальная точка которого лежит вне сферы и который протыкает сферу (протыкает, а не касается ее), имеет с этой сферой две различные точки пересечения. Поэтому в этом случае требуется найти ту точку пересечения, которая находится от начальной точки луча на меньшем расстоянии и отвечает меньшему значению параметра t.

Если же начальная точка луча лежит внутри сферы, то искомая точка пересечения отвечает большему значению параметра t.

Перечислим последовательные шаги описанного алгоритма:

шаг 1-й - найти квадрат расстояния между начальной точкой луча и центром сферы,

шаг 2-й - вычислить квадрат кратчайшего расстояния между лучом и центром сферы,

шаг 3-й - выяснить, лежит ли луч вне сферы,

шаг 4-й - найти разность между квадратом радиуса сферы и квадратом кратчайшего расстояния,

шаг 5-й - выяснить, не является ли это число отрицательным,

шаг 6-й - вычислить расстояние до ближайшей точки пересечения луча со сферой,

шаг 7-й - найти точку пересечения луча со сферой,

шаг 8-й - вычислить единичный вектор нормали сферы в этой точке -

и укажем характер и количество операций на каждом шаге (в предположении, что некоторые из величин вычислены предварительно):

шаг 1-й - 5 сложений или вычитаний и 3 умножения,

шаг 2-й - 2 сложения и 3 умножения,

шаг 3-й - 2 сравнения (1, если начальная точка находится внутри сферы),

шаг 4-й - 2 сложения или вычитания и 1 умножение,

шаг 5-й - 1 сравнение (ни одного, если начальная точка находится внутри сферы),

шаг 6-й - 1 сложение и 1 извлечение квадратного корня,

шаг 7-й - 3 сложения и 3 умножения, шаг

8-й - 3 вычитания и 3 умножения.

Таким образом, предложенный алгоритм отыскания точки пересечения заданного луча и заданной сферы и вычисления единичного

вектора нормали сферы в этой точке требует самое большее 16 сложений или вычитаний, 13 умножений, 1 извлечения квадратного корня и 3 сравнений.

Замечание

Пользователь вправе выбирать тот из описанных выше способов, который кажется ему более простым и естественным, или придумать новый.

2. Пересечение луча с плоскостью

Пусть плоскость задана общим уравнением

$ax + by + cz + d = 0$,

где N = (a, b, c) - нормальный вектор плоскости. В случае, когда вектор N единичный, $a^2 + b^2 + c^2 = 1$, d с точностью до знака равно расстоянию от начала координат (0, 0, 0) до рассматриваемой плоскости. Заменяя в уравнении плоскости величины x, y и z их выражениями (*), получаем линейное уравнение относительно t:

$$a(x_0 + lt) + b(y_0 + mt) + c(z_0 + nt) + d = 0,$$

разрешая которое, находим, что

$$t^* = \frac{-\left(ax_0 + by_0 + cz_0 + d\right)}{al + bm + cn},$$

или, в векторной форме,

$$t^* = \frac{-\left(\left(N \cdot R_0\right) + d\right)}{\left(N \cdot L\right)}.$$

Если скалярное произведение

$$\alpha = \left(N \cdot L\right) = al + bm + cn$$

обращается в нуль,

$$\alpha = \left(N \cdot L\right) = al + bm + cn = 0,$$

то луч параллелен плоскости и, следовательно, не пересекает ее (случай, когда луч лежит в плоскости, практически неинтересен).

Если а ≠ 0, то вычисляем

$$\beta = -\left(\left(N \cdot R_0\right) + d\right)$$

и отношение

$$t^* = \frac{\beta}{\alpha}.$$

В случае

$$t^* < 0$$

луч не пересекает плоскости.

Если

$$\alpha \neq 0 \text{ и } t^* > 0,$$

то координаты точки пересечения вычисляются по формулам

$$x^* = x_0 + lt^*,$$

$$y^* = y_0 + mt^*,$$

$$z^* = z_0 + nt^*.$$

Вектор нормали плоскости в точке ее пересечения с лучом обычно выбирается так, чтобы угол между этим вектором и направляющим вектором луча был тупым, то есть равным

N (в случае, если а < 0) или

- N (в случае, если а > 0) (рис. 12).

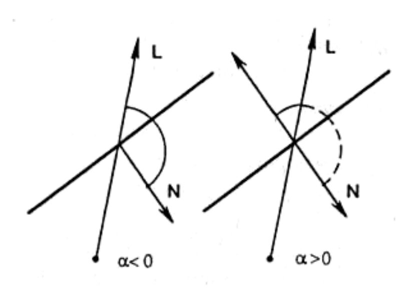


Рис. 12

Перечислим последовательные шаги в описанном алгоритме:

шаг 1-й - вычисление а и сравнение с нулем,

шаг 2-й - вычисление В и t* и сравнение t* с нулем,

шаг 3-й - вычисление точки пересечения луча и плоскости,

шаг 4-й - сравнение а с нулем и (возможное) обращение нормали -и укажем характер и количество операций на каждом шаге:

шаг 1-й - 2 сложения, 3 умножения и 1 сравнение,

шаг 2-й - 3 сложения, 3 умножения и 1 сравнение,

шаг 3-й - 3 сложения и 3 умножения,

шаг 4-й - 1 сравнение.

Таким образом, этот алгоритм требует самое большее 8 сложений или вычитаний, 9 умножений и 3 сравнений.

3. Пересечение луча с выпуклым многоугольником

Выпуклый n-угольник однозначно задается набором своих вершин (x_i, y_i, z_i), i = l,...,n.

Будем считать, что вершины многоугольника занумерованы так, что соседние по номеру вершины примыкают к одной стороне. Обозначим через

(x^*, y^*, z^*)

точку пересечения заданного луча с плоскостью

$ax + by + cz + d = 0$,

в которой лежит рассматриваемый многоугольник, и опишем, как определить, попадает ли эта точка внутрь заданного многоугольника или же лежит вне его.

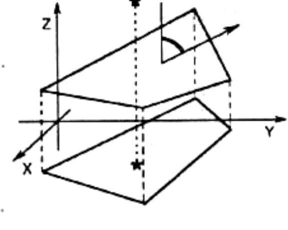


Рис. 13

Для простоты рассуждений случай, когда точка пересечения луча с плоскостью многоугольника попадает на его сторону, из рассмотрения исключим.

Вследствие того, что нормальный вектор N = (a, b, c) плоскости, несущей заданный многоугольник, отличен от нуля, этот n-угольник можно взаимно однозначно спроектировать на n-угольник, лежащий в одной из координатных плоскостей (рис. 13).

Предположим для определенности, что

$c \neq 0$.

Тогда в качестве такой плоскости можно взять координатную плоскость xy, а в качестве направления проектирования - ось аппликат (ось Z).

Рис. 14

Легко видеть, что координаты вершин n-угольника, получающегося в результате такого проектирования, будут иметь вид

(x_i, y_i),

а координаты точки, получающейся в результате проектирования на плоскость xy точки

(x^*, y^*, z^*),

соответственно (x^*, y^*) (рис. 14).

Ясно, что если точка (x^*, y^*) лежит вне (внутри) n-угольника, получившегося на плоскости xy, то исходная точка (x^*, y^*, z^*) будет внешней (внутренней) по отношению к исходному n-угольнику.

Для определения положения точки (x^*, y^*) относительно выпуклого n-угольника, лежащего на плоскости xy, можно поступить, например, следующим образом.

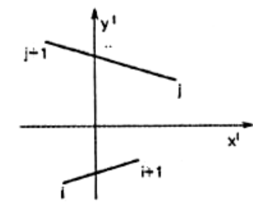


Рис. 15

Передвинем n-угольник

$(x_i, y_i), i = 1,\ldots,n,$

в плоскости xy так, чтобы точка (x^*, y^*) попала в начало координат.

В новой координатной системе абсциссы и ординаты вершин n-угольника будут равны соответственно

$x_i^* = x_i - x^*$ и $y_i^* = y_i - y^*$.

Теперь остается только выяснить, будет (или не будет) точка (0, 0), в которую преобразуется точка (x^*, y^*), внутренней точкой n-угольника с вершинами

$(x_i^*, y_i^*), i = 1,\ldots,n.$

Возможны два случая.

1-й случай.

Абсциссы x_i^* всех вершин n-угольника - одного знака.

Это означает, что рассматриваемая точка лежит вне n-угольника.

2-й случай.

Есть два ребра n-угольника с вершинами

(x_i^*, y_i^*) и (x_{i+1}^*, y_{i+1}^*)

и

(x_j^*, y_j^*) и (x_{j+1}^*, y_{j+1}^*)

соответственно (i < j), такие, что

$x_i^* \cdot x_{i+1}^* < 0$, $x_j^* \cdot x_{j+1}^* < 0$

(рис. 15).

Если

$$\left(y_i^* - \frac{y_{i+1}^* - y_i^*}{x_{i+1}^* - x_i^*}\cdot x_i^*\right)\cdot\left(y_j^* - \frac{y_{j+1}^* - y_j^*}{x_{j+1}^* - x_j^*}\cdot x_j^*\right) < 0,$$

то интересующая нас точка лежит внутри этого n-угольника.

Это означает, что точка (x*, y*, z*) лежит внутри исходного n-угольника.

Если же последнее произведение положительно, то точка (x*, y*, z*) лежит вне исходного n-угольника.

4. Пересечение с прямоугольным параллелепипедом

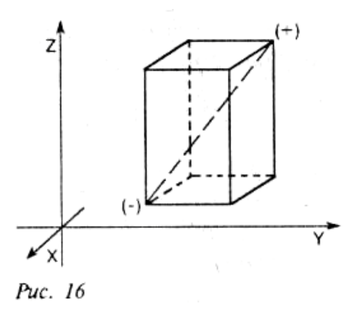
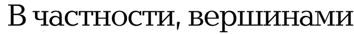


Рис. 16

Прямоугольный параллелепипед со сторонами, параллельными координатным плоскостям, однозначно определяется любыми двумя своими вершинами, примыкающими к одной из диагоналей этого параллелепипеда.


В частности, вершинами

(x_-, y_-, z_-), (x_+, y_+, z_+),

где $x_- < x_+$, $y_- < y_+$, $z_- < z_+$ (рис. 16).

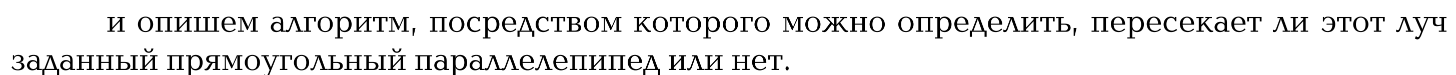
Рассмотрим луч, исходящий из точки

(x_0, y_0, z_0)

в направлении вектора

(l, m, n),

где l2 + m2 + n2 = 1,


и опишем алгоритм, посредством которого можно определить, пересекает ли этот луч заданный прямоугольный параллелепипед или нет.

Противоположные грани рассматриваемого прямоугольного параллелепипеда лежат в плоскостях, параллельных координатным плоскостям. Возьмем, например, пару плоскостей, параллельных плоскости yz:

$x = x_-$ и $x = x_+$

При

$l = 0,$ заданный луч параллелен этим плоскостям и, если

$x_0 < x_-$ или $x_0 > x_+$,

не пересекает рассматриваемый прямоугольный параллелепипед. Если же заданный луч не параллелен этим плоскостям,

$l \neq 0,$

то вычисляем отношения

$$t_{1x} = \frac{x_- - x_0}{1}, t_{2x} = \frac{x_+ - x_0}{1}.$$

Можно считать, что найденные величины связаны неравенством

$t_{1x} < t_{2x}$

(в противном случае просто меняем их местами).

Положим

$t_{near} = t_{1x}$, $t_{far} = t_{2x}$. Считая, что $m \neq 0$,

и рассматривая вторую пару плоскостей, несущих грани заданного параллелепипеда,

$y = y_-$ и $y = y_+$,

находим величины

$$t_{1y} = \frac{y_- - y_0}{m}, t_{2y} = \frac{y_+ - y_0}{m}.$$

Если

$t_{1y} > t_{near}$,

то полагаем

$t_{near} = t_{1y}$

Если

$t_{2y} < t_{far}$, то полагаем

$t_{far} = t_{2y}$.

При

$t_{near} > t_{far}$

или при

$t_{far} < 0$

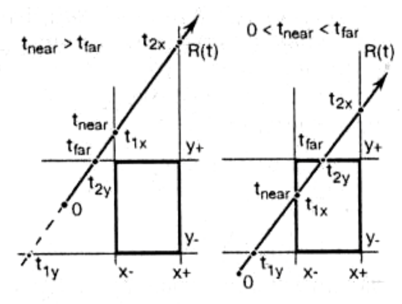


Рис. 17

заданный луч проходит мимо прямоугольного параллелепипеда (рис. 17). Считая

$n \neq 0$,

рассматриваем последнюю пару плоскостей

$z = z_-$ и $z = z_+$,

находим величины

$$t_{1z} = \frac{z_- - z_0}{n}, \; t_{2z} = \frac{z_+ - z_0}{n}$$

и повторяем предыдущие сравнения.

Если в итоге всех проведенных операций мы получим, что

$0 < t_{near} < t_{far}$

или

$0 < t_{far}$,

то заданный луч непременно пересечет исходный параллелепипед со сторонами, параллельными координатным осям.

Если луч протыкает прямоугольный параллелепипед (при этом, естественно, считается, что начальная точка луча лежит вне параллелепипеда), то расстояние от начала луча до точки его входа в параллелепипед равно

t_{near},

а до точки выхода луча из параллелепипеда

t_{far}

соответственно.

Замечание

Рассуждая подобным образом, читатель без особого труда сможет построить алгоритмы отыскания точек пересечения луча с круглым цилиндром -

$$x^2 + y^2 = r^2,$$

конусом -

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = \frac{z^2}{c^2}$$

и другими простыми поверхностями.

УДАЛЕНИЕ НЕВИДИМЫХ ЛИНИЙ И ПОВЕРХНОСТЕЙ

Для построения правильного изображения трехмерных объектов необходимо уметь определять, какие части объектов (ребра, грани) будут видны при заданном проектировании, а какие будут закрыты другими гранями объектов. В качестве возможных видов проектирования традиционно рассматриваются параллельное и центральное (перспективное) проектирования.

Проектирование осуществляется на так называемую картинную плоскость (экран): проектирующий луч к картинной плоскости проводится через каждую точку объектов. При этом видимыми будут те точки, которые вдоль направления проектирования ближе всего расположены к картинной плоскости.

Несмотря на кажущуюся простоту, эта задача является достаточно сложной и требует зачастую больших объемов вычислений. Поэтому существует ряд различных методов решения задач удаления невидимых линий, включая и методы, опирающиеся на аппаратные решения.

Далее будем считать, что все объекты представлены набором выпуклых плоских граней, которые пересекаются только вдоль своих ребер.

К решению задачи удаления невидимых линий и поверхностей можно выделить два основных подхода.

Первый подход заключается в определении для каждого пиксела того объекта, который вдоль направления проектирования является ближайшим к нему. При этом работа ведется в пространстве картинной плоскости и существенно используются растровые свойства дисплея.

Второй подход заключается в непосредственном сравнении объектов друг с другом для выяснения того, какие части каких объектов будут являться видимыми. В данном случае работа ведется в исходном пространстве объектов и никак не привязана к растровым характеристикам дисплея.

Замечание

Существует большое количество смешанных методов, объединяющих оба описанных подхода.

Построение графика функции двух переменных

Рассмотрим сначала задачу построения графика функции двух переменных z = f(x, y) в виде сетки координатных линий x = const и y = const (рис. 1).

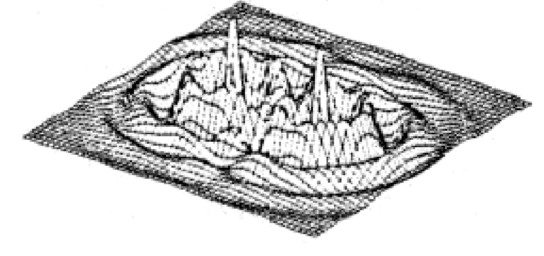


Рис. 1

Будем рассматривать параллельное проектирование, при котором проекцией вертикальной линии на картинной плоскости (экране) является вертикальная линия. Легко убедиться в том, что в этом случае точка p(x, y, z) переходит в точку $((p, e_1), (p, e_2))$ на картинной плоскости, где

$$e_1 = (\cos\varphi, \sin\varphi, 0),$$

$$e_2 = (\sin\varphi\sin\psi, -\cos\varphi\sin\psi, \cos\psi),$$

а направление проектирования имеет вид

$$e_3 = (\sin\varphi\cos\psi, -\cos\varphi\cos\psi, -\sin\psi),$$

где $\varphi \in [0, 2\pi], \psi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

Рассмотрим сначала построение графика функции в виде набора линий, соответствующих постоянным значениям y, считая, что углы φ и ψ подобраны таким образом, что при $y_1 < y_2$ плоскость $y = y_1$ расположена ближе к картинной плоскости, чем плоскость $y = y_2$.

Заметим, что каждая линия семейства $z = f(x, y_i)$ лежит в своей плоскости $y = y_i$, причем все эти плоскости параллельны и, следовательно, не могут пересекаться. Из этого следует, что при $y_j > y_i$ линия $z = f(x, y_j)$ не может закрывать линию $z = f(x, y_i)$.

Тогда возможен следующий алгоритм построения графика функции $z = f(x, y)$: линии рисуются в порядке удаления (возрастания y) и при рисовании очередной линии рисуется только та ее часть, которая не закрывается ранее нарисованными линиями.

Такой алгоритм называется методом плавающего горизонта.

Для определения частей линии $z = f(x, y_k)$, которые не закрывают ранее нарисованные линии, вводятся так называемые линии горизонта, или контурные линии.

Пусть проекцией линии $z = f(x, y_k)$ на картинную плоскость является линия $Y = Y_k(X)$, где (X, Y) - координаты на картинной плоскости, причем Y соответствует вертикальной координате. Контурные линии $Y^k_{\max}(X)$ и $Y^k_{\min}(X)$ определяются следующими соотношениями:

$$Y^k_{\max}(X) = \max_{1 \le i \le k-1} Y_i(X),$$

$$Y^k_{\min}(X) = \min_{1 \le i \le k-1} Y_i(X).$$

На экране рисуются только те части линии $Y = Y_k(X)$, которые находятся выше линии $Y^k_{\max}(X)$ или ниже линии $Y^k_{\min}(X)$.

Одной из наиболее простых и эффективных реализаций данного метода является растровая реализация, при которой в области задания исходной функции вводится сетка

$$\{(x_i, y_j), i = 1,\dots,n_1, j = 1,\dots,n_2\}$$

и каждая из линий $Y = Y_k(X)$ представляется в виде ломаной. Для рисования сегментов этой ломаной используется модифицированный алгоритм Брезенхейма, который перед выводом очередного пиксела сравнивает его ординату с верхней и нижней контурными линиями,

представляющими из себя в этом случае массивы значений ординат.

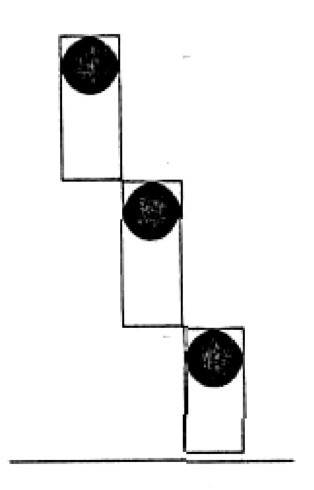


Рис. 2

Замечание

Случай отрезков с угловым коэффициентом большим 1 требует специальной обработки (для того, чтобы не появлялись выпадающие пикселы (рис. 2).

Реализация этого алгоритма приведена на следующем листинге.

```
// File example1.cpp
#include <conio.h>
#include <graphics.h>
#include <math.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

#define NO_VALUE    7777

struct  Point2          // screen point
{
    int   x, y;
};

int     YMax [640];
int     YMin [640];
```

```
int      UpColor   = LIGHTGREEN;
int      DownColor = LIGHTGRAY;

void     DrawLine ( Point2& p1, Point2& p2 )
{
   int   dx = abs ( p2.x - p1.x );
   int   dy = abs ( p2.y - p1.y );
   int   sx = p2.x >= p1.x ? 1 : -1;
   int   sy = p2.y >= p1.y ? 1 : -1;

   if ( dy <= dx )
   {
         int   d  = -dx;
         int   d1 = dy << 1;
         int   d2 = ( dy - dx ) << 1;

         for ( int x = p1.x, y = p1.y, i = 0; i <= dx; i++, x += sx )
         {
               if ( YMin [x] == NO_VALUE ) // YMin, YMax not inited
               {
                     putpixel ( x, y, UpColor );
                     YMin [x] = YMax [x] = y;
               }
               else
               if ( y < YMin [x] )
               {
                     putpixel ( x, y, UpColor );
                     YMin [x] = y;
               }
               else
               if ( y > YMax [x] )
               {
                     putpixel ( x, y, DownColor );
                     YMax [x] = y;
               }

               if ( d > 0 )
               {
                     d += d2;
                     y += sy;
               }
               else
                     d += d1;
         }
   }
   else
   {
         int   d  = -dy;
         int   d1 = dx << 1;
         int   d2 = ( dx - dy ) << 1;
         int   m1 = YMin [p1.x];
         int   m2 = YMax [p1.x];

         for ( int x = p1.x, y = p1.y, i = 0; i <= dy; i++, y += sy )
         {
               if ( YMin [x] == NO_VALUE ) // YMin, YMax not inited
               {
                     putpixel ( x, y, UpColor );
                     YMin [x] = YMax [x] = y;
               }
               else
               if ( y < m1 )
               {
                     putpixel ( x, y, UpColor );
                     if ( y < YMin [x] )
                           YMin [x] = y;
```

```
                }
                else
                if ( y > m2 )
                {
                      putpixel ( x, y, DownColor );
                      if ( y > YMax [x] )
                            YMax [x] = y;
                }

                if ( d > 0 )
                {
                      d += d2;
                      x += sx;
                      m1 = YMin [x];
                      m2 = YMax [x];
                }
                else
                      d += d1;
          }
      }
   }

   void      PlotSurface ( double x1, double y1, double x2, double y2, double (*f)( double,
double ), double fmin, double fmax, int n1, int n2 )
   {
      Point2 * CurLine  = new Point2 [n1];
      double   phi = 30*3.1415926/180;
      double   psi = 10*3.1415926/180;
      double sphi = sin ( phi );
      double cphi = cos ( phi );
      double spsi = sin ( psi );
      double cpsi = cos ( psi );
      double e1 [] = { cphi, sphi, 0 };
      double e2 [] = { spsi*sphi, -spsi*cphi, cpsi };
      double   x, y;
      double hx = ( x2 - x1 ) / n1;
      double hy = ( y2 - y1 ) / n2;
      double   xmin = ( e1 [0] >= 0 ? x1 : x2 )  * e1 [0] + ( e1 [1] >= 0 ? y1 : y2 ) * e1
[1];
      double   xmax = ( e1 [0] >= 0 ? x2 : x1 )  * e1 [0] + ( e1 [1] >= 0 ? y2 : y1 ) * e1
[1];
      double   ymin = ( e2 [0] >= 0 ? x1 : x2 )  * e2 [0] + ( e2 [1] >= 0 ? y1 : y2 ) * e2
[1];
      double   ymax = ( e2 [0] >= 0 ? x2 : x1 )  * e2 [0] + ( e2 [1] >= 0 ? y2 : y1 ) * e2
[1];
      int    i, j, k;

      if ( e2 [2] >= 0 )
      {
            ymin += fmin * e2 [2];
            ymax += fmax * e2 [2];
      }
      else
      {
            ymin += fmax * e2 [2];
            ymax += fmin * e2 [2];
      }

      doubleax = 10 - 600 * xmin / ( xmax - xmin );
      doublebx = 600 / ( xmax - xmin );
      doubleay = 10 - 300 * ymin / ( ymax - ymin );
      doubleby = -300 / ( ymax - ymin );

      for ( i = 0; i < sizeof ( YMax ) / sizeof ( int ); i++ )
            YMin [i] = YMax [i] = NO_VALUE;
```

```
      for ( i = n2 - 1; i > -1; i-- )
      {
            for ( j = 0; j < n1; j++ )
            {
                  x = x1 + j * hx;
                  y = y1 + i * hy;
                  CurLine [j].x = (int)( ax + bx * ( x * e1 [0] + y * e1 [1] ) );
                  CurLine [j].y = (int)( ay + by * ( x * e2 [0] + y * e2 [1] + f ( x, y )
* e2 [2] ) );
            }

            for ( j = 0; j < n1 - 1; j++ )
                  DrawLine ( CurLine [j], CurLine [j + 1] );
      }

      delete CurLine;
   }

   double   f ( double x, double y )
   {
      doubler =  x*x + y*y;
      return cos ( r ) / ( r + 1 );
   }

   main ()
   {
      int   driver = DETECT;
      int   mode;
      int   res;

      initgraph ( &driver, &mode, "" );
      if ( ( res = graphresult () ) != grOk )
      {
            printf("\nGraphics error: %s\n", grapherrormsg ( res) );
            exit ( 1 );
      }

      PlotSurface2 ( -2, -2, 2, 2, f2, -0.5, 1, 40, 40 );

      getch ();
      closegraph ();
   }
```

Для вывода графика, состоящего как из линий $z = f(x, y_i)$, так и из линий $z = f(x_j, y)$, необходимо выводить отрезки ломаных с сохранением порядка загораживания. Ниже приводится текст функции PlotSurface2, осуществляющей такое построение.

```
   // File example2.cpp
   void     PlotSurface2 ( double x1, double y1, double x2, double y2, double (*f)(
double, double ), double fmin, double fmax, int n1, int n2 )
   {
      Point2 * CurLine  = new Point2 [n1];
      Point2 * NextLine = new Point2 [n1];
      double   phi = 30*3.1415926/180;
      double   psi = 10*3.1415926/180;
      double sphi = sin ( phi );
      double cphi = cos ( phi );
      double spsi = sin ( psi );
      double cpsi = cos ( psi );
      double e1 [] = { cphi, sphi, 0 };
      double e2 [] = { spsi*sphi, -spsi*cphi, cpsi };
      double   x, y;
      double hx = ( x2 - x1 ) / n1;
      double hy = ( y2 - y1 ) / n2;
```

```
	double   xmin = ( e1 [0] >= 0 ? x1 : x2 )  * e1 [0] + ( e1 [1] >= 0 ? y1 : y2 ) * e1
[1];
	double   xmax = ( e1 [0] >= 0 ? x2 : x1 )  * e1 [0] + ( e1 [1] >= 0 ? y2 : y1 ) * e1
[1];
	double   ymin = ( e2 [0] >= 0 ? x1 : x2 )  * e2 [0] + ( e2 [1] >= 0 ? y1 : y2 ) * e2
[1];
	double   ymax = ( e2 [0] >= 0 ? x2 : x1 )  * e2 [0] + ( e2 [1] >= 0 ? y2 : y1 ) * e2
[1];
	int    i, j, k;

	if ( e2 [2] >= 0 )
	{
		ymin += fmin * e2 [2];
		ymax += fmax * e2 [2];
	}
	else
	{
		ymin += fmax * e2 [2];
		ymax += fmin * e2 [2];
	}

	doubleax = 10 - 600 * xmin / ( xmax - xmin );
	doublebx = 600 / ( xmax - xmin );
	doubleay = 10 - 400 * ymin / ( ymax - ymin );
	doubleby = -400 / ( ymax - ymin );

	for ( i = 0; i < sizeof ( YMax ) / sizeof ( int ); i++ )
		YMin [i] = YMax [i] = NO_VALUE;

	for ( i = 0; i < n1; i++ )
	{
		x = x1 + i * hx;
		y = y1 + ( n2 - 1 ) * hy;
		CurLine [i].x = (int)( ax + bx * ( x * e1 [0] + y * e1 [1] ) );
		CurLine [i].y = (int)( ay + by * ( x * e2 [0] + y * e2 [1] + f ( x, y ) * e2
[2] ) );
	}

	for ( i = n2 - 1; i > -1; i-- )
	{
		for ( j = 0; j < n1 - 1; j++ )
			DrawLine ( CurLine [j], CurLine [j + 1] );

		if ( i > 0 )
			for ( j = 0; j < n1; j++ )
			{
				x = x1 + j * hx;
				y = y1 + ( i - 1 ) * hy;
				NextLine [j].x = (int)( ax + bx * ( x * e1 [0] + y * e1 [1] ) );
				NextLine [j].y = (int)( ay + by * ( x * e2 [0] + y * e2 [1] + f (
x, y ) * e2 [2] ) );

				DrawLine ( CurLine [j], NextLine [j] );
				CurLine [j] = NextLine [j];
			}
	}

	delete CurLine;
	delete NextLine;
  }
```

Рассмотрим теперь задачу построения полутонового изображения графика функции z = f (x, y). Как и ранее, введем сетку

$$\{(x_i, y_j), i = 1,\dots,n_1, j = 1,\dots,n_2\}$$

и затем приблизим график функции набором треугольных граней с вершинами в точках (x_i, y_j, $f(x_i, y_j)$).

Для удаления невидимых граней воспользуемся аналогичным методом - упорядоченным выводом граней. Только в данном случае треугольники будем выводить не по мере удаления от картинной плоскости, а по мере их приближения, начиная с дальних и заканчивая ближними: треугольники, расположенные ближе к плоскости экрана, выводятся позже и закрывают собой невидимые части более дальних треугольных граней.

Для определения порядка, в котором должны выводиться грани, воспользуемся тем, что треугольники, лежащие в полосе

$\{(x, y), y_i \le y \le y_{i+1}\}$,

не могут закрывать треугольники из полосы

$\{(x, y), y_{i-1} \le y \le y_i\}$.

Приведенная программа реализует этот алгоритм с использованием 256-цветного режима.

```
// File example3.cpp
#include <conio.h>
#include <graphics.h>
#include <math.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

#include "Vector.h"

#define  NO_VALUE    7777

struct   Point2            // screen point
{
   int   x, y;
};

void     PlotShadedSurface ( double x1, double y1, double x2, double y2, double (*f)(
double, double ), double fmin, double fmax, int n1, int n2 )
{
   Point2 * CurLine  = new Point2 [n1];
   Point2 * NextLine = new Point2 [n1];
   Vector * CurPoint = new Vector [n1];
   Vector * NextPoint= new Vector [n1];
   double   phi      = 30*3.1415926/180;
   double   psi      = 20*3.1415926/180;
   double sphi     = sin ( phi );
   double cphi     = cos ( phi );
   double spsi     = sin ( psi );
   double cpsi     = cos ( psi );
   Vector e1 ( cphi, sphi, 0 );
   Vector e2 ( spsi*sphi, -spsi*cphi, cpsi );
   Vector e3 ( sphi*cpsi, -cphi*cpsi, -spsi );
   double   xmin     = ( e1 [0] >= 0 ? x1 : x2 )  * e1 [0] + ( e1 [1] >= 0 ? y1 : y2 )
* e1 [1];
   double   xmax     = ( e1 [0] >= 0 ? x2 : x1 )  * e1 [0] + ( e1 [1] >= 0 ? y2 : y1 )
* e1 [1];
   double   ymin     = ( e2 [0] >= 0 ? x1 : x2 )  * e2 [0] + ( e2 [1] >= 0 ? y1 : y2 )
* e2 [1];
   double   ymax     = ( e2 [0] >= 0 ? x2 : x1 )  * e2 [0] + ( e2 [1] >= 0 ? y2 : y1 )
* e2 [1];
   double hx       = ( x2 - x1 ) / n1;
   double hy       = ( y2 - y1 ) / n2;
   Vector Edge1, Edge2, n;
   Point2 facet [3];
   double   x, y;
```

```
int    color;
int    i, j, k;

if ( e2 [2] >= 0 )
{
      ymin += fmin * e2 [2];
      ymax += fmax * e2 [2];
}
else
{
      ymin += fmax * e2 [2];
      ymax += fmin * e2 [2];
}

doubleax = 20 - 600 * xmin / ( xmax - xmin );
doublebx = 600 / ( xmax - xmin );
doubleay = 40 - 400 * ymin / ( ymax - ymin );
doubleby = -400 / ( ymax - ymin );

for ( i = 0; i < 64; i++ )
{
      setrgbpalette ( i,      0, 0, i );
      setrgbpalette ( 64 + i, 0, i, 0 );
     }

for ( i = 0; i < n1; i++ )
{
      CurPoint [i].x = x1 + i * hx;
      CurPoint [i].y = y1;
      CurPoint [i].z = f ( CurPoint [i].x, CurPoint [i].y );
      CurLine [i].x = (int)( ax + bx * ( CurPoint [i] & e1 ) );
      CurLine [i].y = (int)( ay + by * ( CurPoint [i] & e2 ) );
}

for ( i = 1; i < n2; i++ )
{
      for ( j = 0; j < n1; j++ )
      {
            NextPoint [j].x = x1 + j * hx;
            NextPoint [j].y = y1 + i * hy;
            NextPoint [j].z = f ( NextPoint [j].x, NextPoint [j].y );
            NextLine [j].x = (int)( ax + bx * ( NextPoint [j] & e1 ) );
            NextLine [j].y = (int)( ay + by * ( NextPoint [j] & e2 ) );
      }

      for ( j = 0; j < n1 - 1; j++ )
      {
                              // draw 1st triangle
            Edge1 = CurPoint [j+1] - CurPoint [j];
            Edge2 = NextPoint [j]  - CurPoint [j];
            n     = Edge1 ^ Edge2;

            if ( ( n & e3 ) >= 0 )
                  color = 64 + (int)( 20 + 43 * ( n & e3 ) / !n );
            else
                  color = (int)( 20 - 43 * ( n & e3 ) / !n );

            setfillstyle ( SOLID_FILL, color );
            setcolor ( color );

            facet [0] = CurLine [j];
            facet [1] = CurLine [j+1];
            facet [2] = NextLine [j];

            fillpoly ( 3, ( int far * ) facet );
```

```
                              // draw 2nd triangle
              Edge1 = NextPoint [j+1] - CurPoint [j+1];
              Edge2 = NextPoint [j]   - CurPoint [j+1];
              n     = Edge1 ^ Edge2;

              if ( ( n & e3 ) >= 0 )
              {
                     color = 127;
                     color = 64 + (int)( 20 + 43 * ( n & e3 ) / !n );
              }
              else
              {
                     color = 63;
                     color = (int)( 20 - 43 * ( n & e3 ) / !n );

              }

              setfillstyle ( SOLID_FILL, color );
              setcolor ( color );

              facet [0] = CurLine  [j+1];
              facet [1] = NextLine [j];
              facet [2] = NextLine [j+1];

              fillpoly ( 3, ( int far * ) facet );
       }

       for ( j = 0; j < n1; j++ )
       {
              CurLine  [j] = NextLine [j];
              CurPoint [j] = NextPoint [j];
       }
   }

   delete CurLine;
   delete NextLine;
   delete CurPoint;
   delete NextPoint;
}

double   f2 ( double x, double y )
{
   doubler =  x*x + y*y;
   return cos ( r ) / ( r + 1 );
}


main ()
{
   int   driver;
   int   mode;
   int   res;

   if ( ( driver = installuserdriver ( "VESA", NULL ) ) == grError )
   {
         printf ( "\nCannot load extended driver" );
         exit ( 1 );
   }

   initgraph ( &driver, &mode, "" );
   if ( ( res = graphresult () ) != grOk )
   {
         printf("\nGraphics error: %s\n", grapherrormsg ( res) );
         exit ( 1 );
   }
```

```
    PlotShadedSurface ( -2, -2, 2, 2, f2, -0.5, 1, 30, 30 );

    getch ();
    closegraph ();
}
```

Отсечение нелицевых граней

Рассмотрим многогранник, для каждой грани которого задан единичный вектор внешней нормали (рис. 3). Несложно заметить, что если вектор нормали грани n составляет с вектором 1, задающим направление проектирования, тупой угол, то эта грань заведомо не может быть видна. Такие грани называются нелицевыми. В случае, когда соответствующий угол является острым, грань называется лицевой.

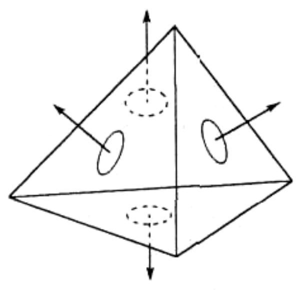


Рис. 3

В случае параллельного проектирования условия на угол можно записать в виде

$(n, 1) =< 0$, поскольку направление проектирования 1 от грани не зависит.

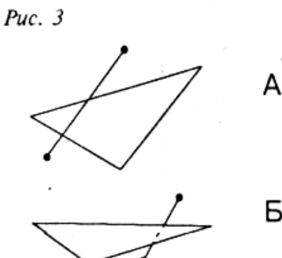
При центральном проектировании с центром в точке с вектор проектирования для точки p будет равен

$1 = c - p$.

Для определения того, является заданная грань лицевой или нет, достаточно взять произвольную точку p этой грани и проверить выполнение условия

$(n, 1) \leq 0$.

Знак этого скалярного произведения не зависит от выбора точки грани, а определяется тем, в каком полупространстве относительно плоскости, содержащей данную грань, лежит центр проектирования.



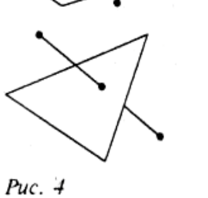
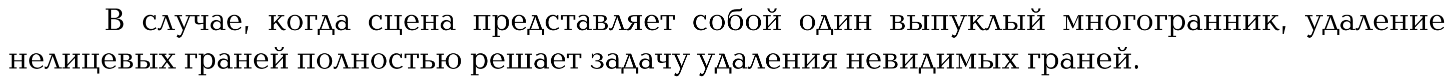


Рис. 4



В случае, когда сцена представляет собой один выпуклый многогранник, удаление нелицевых граней полностью решает задачу удаления невидимых граней.

В общем случае предложенный подход хотя и не решает задачу полностью, но позволяет примерно вдвое сократить количество рассматриваемых граней.

Удаление невидимых линий. Алгоритм Робертса

Самым первым алгоритмом, предназначенным для удаления невидимых линий, был алгоритм Робертса, требующий, чтобы каждая грань была выпуклым многогранником.

Опишем этот алгоритм.

Сначала отбрасываются все ребра, обе определяющие грани которых являются нелицевыми (ни одно из таких ребер не будет видно).

Следующим шагом является проверка каждого из оставшихся ребер со всеми гранями многогранника на закрывание. Возможны следующие случаи (рис. 4):

- грань не закрывает ребро;
- грань полностью закрывает ребро (и оно тогда удаляется из списка рассматриваемых ребер);
- грань частично закрывает ребро (в этом случае ребро разбивается на несколько частей, из которых видимыми являются не более двух; само ребро удаляется из списка, но в этот список проверенных ребер добавляются те его части, которые не закрываются данной гранью). Если общее количество граней равно n, то временные затраты для данного алгоритма составляют O(n2).

Можно заметно сократить количество проверок, если воспользоваться разбиением картинной плоскости.

Картинная плоскость разбивается на равные клетки, и для каждой клетки составляется список тех граней, проекции которых имеют непустое пересечение с данной клеткой (рис. 5). Для проверки произвольного ребра сначала находятся все клетки, в которые попадает проекция этого ребра, и рассматриваются только те грани, которые содержатся в списках данных клеток.

Несмотря на то, что этот вариант алгоритма требует определенных затрат для построения разбиения и соответствующих списков, при удачном выборе разбиения он имеет порядок 0(n).

Рис. 5

Алгоритм Аппеля

Введем понятие так называемой количественной невидимости (quontative invisibility) точки как количества лицевых граней, ее закрывающих.

Точка является видимой только в том случае, когда ее количественная невидимость равна нулю.

Рассмотрим, как меняется количественная невидимость вдоль ребра.

Количественная невидимость точек ребра изменяется на единицу при прохождении ребра позади так называемой контурной линии, состоящей из тех ребер, для которых одна из проходящих граней является лицевой, а другая - нелицевой.

Так, для многогранника на рис. 6 контурной линией является ломаная ABCIJDEKLGA.

Для определения видимости ребер произвольного многогранника берется какая-нибудь его вершина и затем непосредственно определяется ее количественная невидимость.

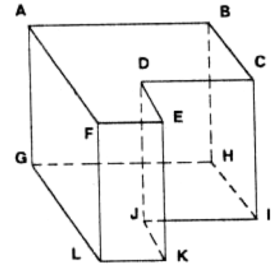


Рис. 6

Далее прослеживается изменение количественной невидимости вдоль каждого из ребер, выходящих из этой вершины. Эти ребра проверяются на прохождение позади контурной линии, и в соответствующих точках их количественная невидимость изменяется. Те части отрезка, для которых количественная невидимость равна нулю, сразу рисуются.

Следующим шагом является определение количественной невидимости для всех ребер, выходящих из новой вершины, и т. д.

В результате определяется количественная невидимость связной компоненты сцены, содержащей исходную вершину (и при этом она рисуется).

В случае, когда рассматривается изменение количественной невидимости вдоль ребра, выходящего из вершины, принадлежащей контурной линии, необходимо проверить, не закрывается ли это ребро одной из граней, выходящей из этой вершины, как, например, грань DEKJ закрывает ребро DJ.

Так как для реальных объектов количество ребер, входящих в контурную линию, намного меньше общего числа ребер, то алгоритм Аппеля является более эффективным, чем алгоритм Робертса.

Замечание

Для повышения эффективности данного алгоритма также возможно использование разбиения картинной плоскости.

Удаление невидимых граней. Метод z-буфера

Одним из самых простых алгоритмов удаления невидимых граней и поверхностей является метод z-буфера (буфера глубины). В силу крайней простоты этого метода часто встречаются его аппаратные реализации.

Сопоставим каждому пикселу (x, y) картинной плоскости, кроме цвета, хранящегося в видеопамяти, его расстояние до картинной плоскости вдоль направления проектирования z(x, y) (его глубину).

Изначально массив глубин инициализируется $+\infty$.

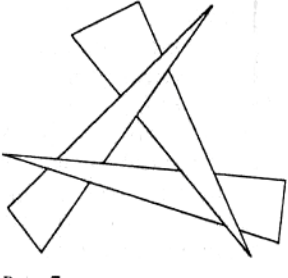
Для вывода на картинную плоскость произвольной грани она переводится в свое растровое представление на картинной плоскости и для каждого пиксела этой грани находится его глубина. В случае, если эта глубина меньше значения глубины, хранящегося в z-буфере, пиксел рисуется и его глубина заносится в z-буфер.

Замечание

Данный метод работает исключительно в пространстве картинной плоскости и не требует никакой предварительной обработки данных. Для вычисления глубины соседних пикселов при растровом разложении грани может использоваться вариант целочисленного алгоритма Брезенхейма.

Алгоритмы упорядочения

Метод, использованный ранее для построения графика функции двух переменных и заключающийся в последовательном выводе на экран всех граней в определенном порядке, может быть использован и для расчета более сложных сцен.


Рис. 7

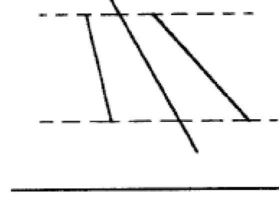
Подход заключается в таком упорядочении граней, чтобы при их выводе в этом порядке получалось корректное изображение. Для этого необходимо, чтобы более дальние грани выводились раньше, чем более близкие.

Существуют различные методы построения такого упорядочения, однако часто встречаются такие случаи, когда заданные грани нельзя упорядочить (рис. 7). В подобных случаях необходимо произвести разбиение одной или нескольких граней, чтобы получившееся после разбиения множество граней можно было упорядочить.

Метод сортировки по глубине

Наиболее простым подходом к упорядочиванию граней является их сортировка по минимальному расстоянию до картинной плоскости (вдоль направления проектирования) с последующим выводом их в порядке приближения.

Этот метод великолепно работает для ряда сцен, включая, например, построение изображения нескольких непересекающихся достаточно простых тел.


Рис. 8

Однако возможны случаи, когда просто сортировка по расстоянию до картинной плоскости не обеспечивает правильного упорядочения граней (рис. 8); поэтому желательно после такой сортировки проверить порядок, в котором грани будут выводиться.

Предлагается следующий алгоритм этой проверки. Для простоты будем считать, что рассматривается параллельное проектирование вдоль оси Oz.

Перед выводом грани P следует убедиться, что никакая другая грань Q, проекция которой на ось Oz пересекается с проекцией грани P, не может закрываться гранью P. И если это условие выполнено, то грань P должна быть выведена раньше. Предлагаются следующие 5 тестов в порядке возрастания сложности проверки:

1. Пересекаются ли проекции этих граней на ось Ox?
2. Пересекаются ли их проекции на ось Oy?

3. Находится ли грань P по другую сторону от плоскости, проходящей через грань Q, чем начало координат (наблюдатель)?

4. Находится ли грань Q по ту же сторону от плоскости, проходящей через грань P, что и начало координат (наблюдатель)?

5. Пересекаются ли проекции этих граней на картинную плоскость?

Если хотя бы на один из этих вопросов получен отрицательный ответ, то считаем что эти две грани - P и Q упорядочены верно, и сравниваем P со следующей гранью. В противном случае считаем, что эти грани необходимо поменять местами, для чего проверяются следующие тесты:

3'. Находится ли грань Q по другую сторону от плоскости, проходящей через грань P, чем начало координат?

4'. Находится ли грань P по ту же сторону от плоскости, проходящей через грань Q, что и начало координат?

В случае если ни один из этих тестов не позволяет с уверенностью решить, какую из этих двух граней нужно выводить раньше, то одна из них разбивается плоскостью, проходящей через другую грань. В этом случае вопрос об упорядочении оставшейся грани и частей разбитой грани легко решается.

Метод двоичного разбиения пространства

Существует другой, крайне элегантный способ упорядочивания граней.

Рассмотрим некоторую плоскость в объектном пространстве. Она разбивает множество всех граней на два непересекающихся множества (кластера), в зависимости от того, в каком полупространстве относительно плоскости эти грани лежат (будем считать, что плоскость не пересекает ни одну из этих граней).

При этом очевидно, что ни одна из граней, лежащих в полупространстве, не содержащем наблюдателя, не может закрывать собой ни одну из граней, лежащих в том же полупространстве, что и наблюдатель. Тем самым сначала необходимо вывести грани из дальнего кластера, а затем уже и из ближнего.

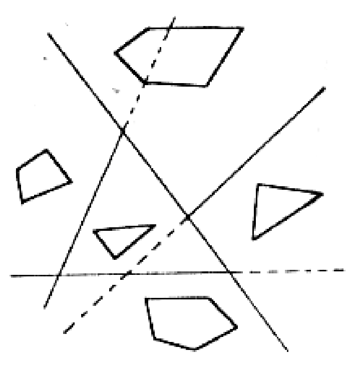


Рис. 9

Применим подобную технику для упорядочения граней внутри каждого кластера. Для этого для построим разбиение граней каждого кластера на два множества очередной плоскостью; а затем для вновь полученных граней повторим процесс разбиения, и будем поступать так, до тех пор, пока в каждом получившемся кластере останется не более одной грани (рис. 9).

Обычно в качестве разбивающей плоскости рассматривается плоскость, проходящая через одну из граней (на самом деле при этом множество всех граней разбивается на 4 класса - лежащих на плоскости, пересекающих ее, лежащих в положительном полупространстве и лежащие в отрицательном полупространстве относительно этой плоскости). Все грани, пересекаемые плоскостью, разобьем вдоль этой плоскости.

В результате мы приходим к дереву разбиения пространства (Binary Space Partitioning), узлами которого являются грани. Каждый узел такого дерева можно представить в виде следующей структуры:

```
struct BSPNode
{
    Facet * facet;    // corresponding facet
    Vector n;   // normal to facet ( plane )
    double d;   // plane parameter
    BSPNode * Left;   // left subtree
    BSPNode * Right;  // right subtree
```

```
};
```

При этом Left указывает на вершину поддерева, содержащуюся в положительном полупространстве (p, n) > d, а Right - на поддерево, содержащееся в отрицательном полупространстве (p, n) < d.

Процесс построения дерева заключается в выборе грани, проведении через нее плоскости и разбиении множества всех граней. В этом процессе присутствует определенный произвол в выборе очередной грани. Существует два основных критерия для выбора:

- получить как можно более сбалансированное дерево;
- минимизировать количество разбиений.

К сожалению, эти критерии, как правило, являются взаимоисключающими, поэтому выбирается некоторый компромиссный вариант.

После того как это дерево построено, осуществляется построение изображения в зависимости от используемого проектирования. Ниже приводится процедура построения изображения для центрального проектирования с центром в точке c.

```
void DrawBSPTree ( BSPNode * Tree )
{
    if ( ( Tree -> n & c ) > Tree -> d )
    {
       if ( Tree -> Right != NULL )
          DrawBSPTree ( Tree -> Right );
       DrawFacet ( Tree -> facet );
       if ( Tree -> Left != NULL )
          DrawBSPTree ( Tree -> Left );
    }
    else
    {
       if ( Tree -> Left != NULL )
          DrawBSPTree ( Tree -> Left );
       DrawFacet ( Tree -> facet );
       if ( Tree -> Right != NULL )
          DrawBSPTree ( Tree -> Right );
    }
}
```

Одним из основных преимуществ этого метода является его полная независимость от положения центра проектирования, что делает его крайне удобным для построения серий изображений одной и той же сцены из разных точек наблюдения.

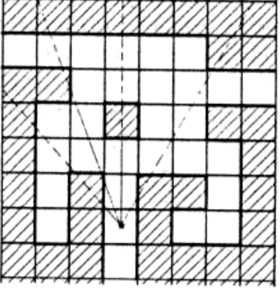
Метод построчного сканирования

Метод построчного сканирования является еще одним примером метода, работающего в пространстве картинной плоскости. Однако вместо того, чтобы решать задачу удаления невидимых граней для проекций объектов на картинную плоскость, сведем ее к серии простых одномерных задач. Все изображение на картинной плоскости можно представить как ряд горизонтальных (вертикальных) линий пикселов. Рассмотрим сечение сцены плоскостью, проходящей через такую линию пикселов и центр проектирования. Пересечением этой плоскости с объектами сцены будет множество непересекающихся (за исключением концов) отрезков, которые и необходимо спроектировать. Задача удаления невидимых частей для такого набора отрезков решается тривиально. Рассматривая задачу удаления невидимых граней для каждой такой линии, мы тем самым разбиваем исходную задачу на набор гораздо более простых задач.

Подобные алгоритмы с успехом используются для создания компьютерных игр типа Wolfenstein 3d и DOOM.

Рассмотрим, каким путем возможно применение этого метода для создания игры типа Wolfenstein 3d.

В этой игре вся сцена представляет собой прямоугольный лабиринт с постоянной высотой пола и потолка и набором вертикальных стен (рис. 10, вид сверху).


Рис. 10

Разложим изображение сцены в ряд вертикальных линий. Каждая такая линия однозначно определяет вертикальную полуплоскость, проходящую через нее и точку наблюдения. Ясно, что в данном случае среди всех пересечений этой полуплоскости со стенами лабиринта, видимым будет только одно, ближайшее. При рассматриваемых условиях вся задача поиска пересечений может решаться в плоскости Оху, что позволяет свести ее к поиску пересечений луча с набором отрезков, представляющих собой проекции стен лабиринта.

После того, как такое пересечение построено, пользуясь свойствами центрального проектирования, находится проекция стены на эту линию.

На самом деле каждая вертикальная линия изображения состоит из трех частей - пола, части стены и потолка. Поэтому после определения части линии, занимаемой проекцией стены (она представляет собой отрезок), оставшаяся часть линии заполняется цветом пола и потолка.

Алгоритм Варнака

Алгоритм Варнака является еще одним примером алгоритма, основанного на разбиении картинной плоскости на части, для каждой из которых исходная задача может быть решена достаточно просто.

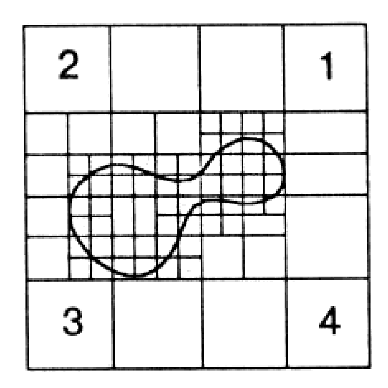


Рис. 11

Разобьем видимую часть картинной плоскости на 4 равные части. В случаях, когда часть полностью накрывается проекцией ближайшей грани и часть не накрывается проекцией ни одной грани вопрос о закрашивании соответствующей части решается тривиально.

В случае, когда ни одно из этих условий не выполнено, данная часть разбивается на 4 части, для каждой из которых проверяется выполнение этих условий, и так далее. Очевидно, что разбиение имеет смысл проводить до тех пор, пока размер части больше чем размер пиксела. В противном случае для части размером в один пиксел явно находится ближайшая к ней грань и осуществляется закрашивание (рис. 11).

ЗАКРАШИВАНИЕ

Следующим шагом на пути создания реалистических изображений является проблема закрашивания поверхностей, ограничивающих построенные объекты. В этой главе мы остановимся на описании некоторых простейших моделей, требующих сравнительно небольших вычислительных затрат. Описанию более совершенных методов построения реалистических изображений - метода трассировки лучей и метода излучательности - посвящены заключительные главы этой части.

Световая энергия, падающая на поверхность от источника света, может быть поглощена, отражена и пропущена. Количество поглощенной, отраженной и пропущенной энергии зависит от длины световой волны. При этом цвет поверхности объекта определяется поглощаемыми длинами волн.

Свойства отраженного света зависят от формы и направления источника света, а также от ориентации освещаемой поверхности и ее свойств. Свет, отраженный от объекта, может быть диффузным и зеркальным: диффузно отраженный свет рассеивается равномерно по всем направлениям, зеркальное отражение происходит от внешней поверхности объекта.

Свет точечного источника отражается от идеального рассеивателя по закону косинусов Ламберта:

$$I = I_l k_d \cos\Theta, 0 \le \Theta \le \frac{\pi}{2}, \qquad (1)$$

где I - интенсивность отраженного света; I_l - интенсивность точечного источника; k_d - коэффициент диффузного отражения (постоянная величина, $0 \le k_d \le 1$); θ - угол между направлением на источник света и (внешней) нормалью к поверхности (рис. 1).

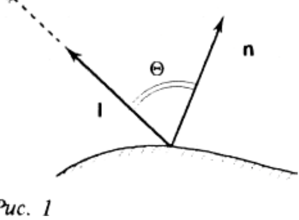


Рис. 1

На объекты реальных сцен падает еще и рассеянный свет, соответствующий отражению света от других объектов. Поскольку точный расчет рассеянного освещения требует значительных вычислительных затрат, в компьютерной графике при вычислении интенсивности поступают так :

$$I = I_a k_a + I_1 k_d \cos\Theta, 0 \le \Theta \le \frac{\pi}{2}, \qquad (2)$$

где I_a - интенсивность рассеянного света;

k_a - (постоянный) коэффициент диффузного отражения рассеянного света, $0 \le k_a \le 1$.

Интенсивность света, естественно, зависит от расстояния d от объекта до источника света. Для того чтобы учесть это, пользуются следующей моделью освещения:

$$I = I_a k_a + \frac{I_1 k_d}{d + K} \cos\Theta, \qquad (3)$$

где K - произвольная постоянная.

Интенсивность зеркально отраженного света зависит от угла падения, длины волны и свойств вещества. Так как физические свойства зеркального отражения довольно сложны, то в простых моделях освещения обычно пользуются следующей эмпирической моделью (моделью Фонга):

$$I_S = I_1 k_S \cos^P \alpha, \qquad (4)$$

Рис. 2

где k_s - экспериментальная постоянная;

α - угол между отраженным лучом и вектором наблюдения;

p - степень, аппроксимирующая пространственное распределение света (рис. 2).

Объединяя последние две формулы,

получаем модель освещения (функцию закраски), используемую для расчета интенсивности (или тона) точек поверхности объекта (или пикселов изображения):

$$I = I_a k_a + \frac{I_1}{d+K}(k_d \cos\Theta + k_S \cos^p \alpha). \qquad (5)$$

Функцию закраски, используя единичные векторы внешней нормали n, а также единичные векторы, определяющие направления: на источник (вектор 1), отраженного луча (вектор r) и наблюдения (вектор s), можно записать в следующем виде:

$$I = I_a k_a \frac{I_1}{d+K}\left(k_d (n \cdot 1) + k_S (r \cdot s)^p\right). \qquad (6)$$

Замечание

Чтобы получить цветное изображение, необходимо найти функции закраски для каждого из трех основных цветов - красного, зеленого и синего. Поскольку цвет зеркально отраженного света определяется цветом падающего, то постоянная k_s считается одинаковой для каждого из этих цветов.

Если точечных источников света несколько, скажем т, то модель освещения определяется так

$$I = I_a k_a + \sum_{j=1}^{m} \frac{I_{1_j}}{d+K}\left(k_d \cos\Theta_j + k_S \cos^{p_j} \alpha_j\right). \qquad (7)$$

Если освещаемая поверхность в рассматриваемой точке гладкая (имеет касательную плоскость), то вектор внешней нормали вычисляется непосредственно (рис. 3). В случае многогранной поверхности векторы внешних нормалей можно найти только для ее граней. Что касается направлений векторов внешних нормалей на ребрах и в вершинах этой поверхности, то их значения можно найти только приближенно.

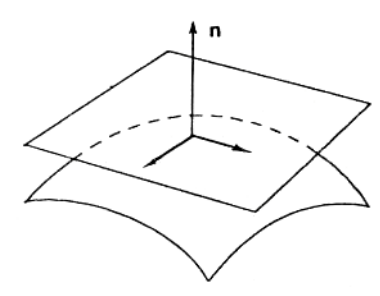


Рис. 3

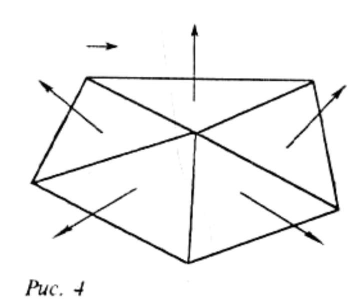
Пусть, например, грани, сходящиеся в данной вершине, лежат в плоскостях, описываемых уравнениями

$$A_i x + B_i y + C_i z + D_i = 0, i = 1,\ldots,m.$$

Можно считать, что нормальные векторы этих плоскостей

$$(A_i, B_i, C_i), i = 1,\ldots,m,$$

являются векторами внешних нормалей для рассматриваемой многогранной поверхности (если какой-то из нормальных векторов не является внешним, то достаточно поменять знаки его координат на противоположные) (рис. 4). Складывая эти векторы, получаем вектор, определяющий направление приближенной нормали


Рис. 4

$$(A, B, C) = \sum_{i=1}^{m}(A_i, B_i, C_i).$$

Замечание

Для определения направления приближенной нормали в точке, лежащей на ребре многогранной поверхности, достаточно сложить векторы внешних нормалей, примыкающих к этому ребру граней рассматриваемой поверхности (рис. 5). Можно поступить и по-иному. А именно, аппроксимировать переменный вектор нормали вдоль ребра многогранной поверхности при помощи уже найденных векторов внешних нормалей в вершинах, прилегающих к рассматриваемому ребру.

Рис. 5

Пусть многогранная поверхность задана своими вершинами (рис. 6). Тогда векторы, определяющие направления приближенных внешних нормалей в ее вершинах можно найти, используя векторные произведения, построенные на векторах, идущих вдоль ребер, исходящих из соответствующих вершин. Например, для того,

Рис. 6

чтобы определить внешнюю нормаль в вершине V_1, необходимо сложить векторные произведения

V_1V_2 x V_1V_3 , V_1V_3 x V_1V_4 ,

V_1V_4 x V_1V_2 .

Замечание

Если перед сложением найденные векторные произведения пронормировать, то полученная сумма будет отличаться от предыдущей и по длине, и по направлению.

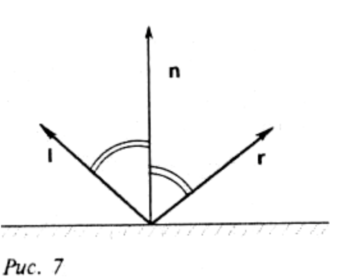


Рис. 7

Для отыскания направления вектора отражения напомним, что единичные векторы - падающего света 1, нормали к поверхности n и отражения r лежат в одной плоскости, причем угол падения равен углу отражения (рис. 7).

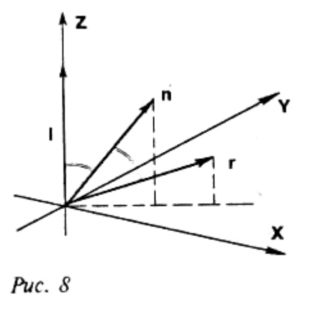


Рис. 8

Рассмотрим модель освещения с одним точечным источником и предположим, что свет падает вдоль оси Z (рис. 8). Тогда координаты единичного вектора отражения

$r = (r_x, r_y, r_z)$

определяются по формулам

$r_x = 2n_x n_z$,

$r_y = 2n_y n_z$,

$r_z = 2n_z^2 - 1$,

где n = (n_x, n_y, n_z) - единичный вектор нормали к поверхности.

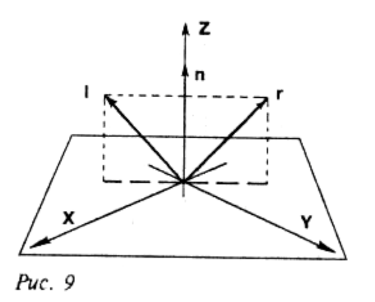


Рис. 9

Если же свет от источника падает не по оси аппликат, то проще всего

поступить так: выбрать новую координатную систему так, чтобы ее начало совпадало с рассматриваемой точкой, касательная плоскость к поверхности была плоскостью xy, а нормаль к поверхности в этой точке шла вдоль оси Z (рис. 9). В этой новой системе координаты векторов r и 1 будут связаны соотношениями

$r_x = -1_x, r_y = -1_y, r_z = 1_z$.

Для того чтобы получить исходные координаты вектора отражения, необходимо выполнить обратное преобразование.

Рассмотрим произвольную сцену, составленную из полигональных (многогранных) фигур. Простейший способ ее построения заключается в том, что на каждой из граней выбирается по точке, для нее определяется освещенность, и затем вся грань закрашивается с использованием найденной освещенности. Предложенный алгоритм обладает, однако, одним большим недостатком - полученное изображение имеет неестественный многогранный вид. Это объясняется тем, что определяемая подобным образом освещенность сцены не является непрерывной величиной, а имеет кусочно-постоянный характер.

Существуют специальные методы закрашивания, позволяющие создавать иллюзию гладкости.

Опишем два известных метода построения сглаженных изображений.

Закраска методом Гуро

Наиболее простым из таких методов является метод Гуро, который основывается на определении освещенности грани в ее вершинах с последующей билинейной интерполяцией получившихся величин на всю грань.

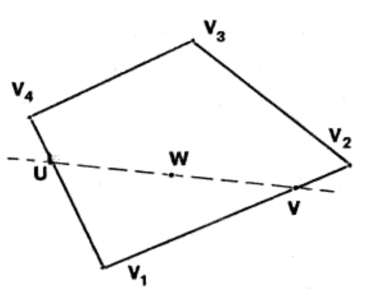


Рис. 10

Обратимся к рис. 10, на котором изображена выпуклая четырехугольная грань. Предположим, что интенсивности в ее вершинах V_1, V_2, V_3 и V_4 известны и равны соответственно I_{v1}, I_{v2}, I_{V3} и I_{v4}.

Пусть W - произвольная точка грани. Для определения интенсивности (освещенности) в этой точке проведем через нее горизонтальную прямую. Обозначим через U и V точки пересечения проведенной прямой с границей грани.

Будем считать, что интенсивность на отрезке UV изменяется линейно, то есть

$I_W = (1-t)I_U + tI_V$,

где $t = \dfrac{|UW|}{|UV|}, 0 \le t \le 1.$

Для определения интенсивности в точках U и V вновь воспользуемся линейной интерполяцией, также считая, что вдоль каждого из ребер границы интенсивность изменяется линейно.

Тогда интенсивность в точках U и V вычисляется по формулам

$$I_U = (1-u)I_{V_4} + uI_{V_1},$$

$$I_V = (1-v)I_{V_1} + vI_{V_2},$$

где $u = \dfrac{|V_4U|}{|V_4V_1|}, 0 \le u \le 1, v = \dfrac{|V_1V|}{|V_1V_2|}, 0 \le v \le 1.$

Метод Гуро обеспечивает непрерывное изменение интенсивности при переходе от одной грани к другой без разрывов и скачков.

Еще одним преимуществом этого метода является его инкрементальный характер: грань рисуется в виде набора горизонтальных отрезков, причем так, что интенсивность последующего пиксела отрезка отличается от интенсивности предыдущего на величину постоянную для данного отрезка. Кроме того, при переходе от отрезка к отрезку значения интенсивности в его концах также изменяются линейно.

Таким образом, процесс рисования грани слагается из следующих шагов:

1) проектирование вершин грани на экран;

2) отыскание интенсивностей в вершинах по формуле (3);

3) определение координат концов очередного отрезка и значений интенсивности в них линейной интерполяцией;

4) рисование отрезка с линейным изменением интенсивности между его концами.

Замечания:

1. При определении освещенности в вершине, естественно, встает вопрос о выборе нормали. Часто в качестве нормали в вершине выбирается нормированная сумма нормалей прилегающих граней

$$n = \frac{a_1 n_1 + ... + a_k n_k}{|a_1 n_1 + ... + a_k n_k|},$$

где $a_1,..., a_k$ - произвольные весовые коэффициенты.

2. Дефекты изображения, возникающие при закраске Гуро, частично объясняются тем, что этот метод не обеспечивает гладкости изменения интенсивности.

Закраска методом Фонга

Как и описанный выше метод закраски Гуро, закраска Фонга при расчете интенсивности также опирается на интерполирование. Однако в отличие от метода Гуро здесь

интерполируется не значение интенсивности по уже известным ее значениям в опорных точках, а значение вектора внешней нормали, которое затем используется для вычисления интенсивности пиксела. Поэтому закраска Фонга требует заметно большего объема вычислений. Правда, при этом и изображение получается более близким к реалистичному (в частности, при закраске Фонга зеркальные блики выглядят довольно правдоподобно).

Метод Фонга заключается в построении для каждой точки вектора, играющего роль вектора внешней нормали, и использовании этого вектора для вычисления освещенности в рассматриваемой точке по формуле (5). При этом схема интерполяции, используемая при закраске Фонга, аналогична интерполяции в закраске Гуро.

Для определения вектора "нормали" n_w в точке W проводим через эту точку горизонтальную прямую и, используя значения векторов "нормалей" n_U и n_v в точках ее пересечения U и V с ребрами грани, получаем

$$n_W = \frac{(1-t)n_U + tn_V}{\left|(1-t)n_U + tn_V\right|},$$

где $t = \dfrac{|UW|}{|UV|}$,

а векторы внешних нормалей в точках U и V находятся, в свою очередь (также линейной интерполяцией), по векторам нормалей в концевых точках соответствующих ребер рассматриваемой многоугольной грани:

$$n_U = (1-u)n_{V_4} + un_{V_1},$$

$$n_V = (1-v)n_{V_1} + vn_{V_2},$$

где $u = \dfrac{|V_4U|}{|V_4V_1|}, v = \dfrac{|V_1V|}{|V_1V_2|}$.

Нормирование вектора n_w необходимо в следствие того, что в формулах (1)-(5) используется единичный вектор нормали.

Замечания:

1. Как и метод Гуро метод Фонга также в значительной степени носит инкрементальный характер.

2. Применяя метод Фонга, мы фактически строим на многогранной модели непрерывное поле единичных векторов, использование которого в качестве поля внешних нормалей обеспечивает гладкость получаемого изображения.

3. Ясно, что требования к качеству изображения напрямую связаны с точностью рассматриваемой модели и объемом соответствующих ей вычислений. Несомненным достоинством предложенных моделей закраски (Гуро и Фонга) является их сравнительная простота. Однако вследствие значительных упрощений получаемый результат не всегда оказывается удовлетворительным. Преодолевать этот . барьер качества лучше всего путем использования более совершенных моделей и методов. Описанию именно таких методов и посвящены заключительные главы настоящей части.

ГЕОМЕТРИЧЕСКИЕ СПЛАЙНЫ

Историю сплайнов принято отсчитывать от момента появления первой работы Шенберга в 1946 году. Сначала сплайны рассматривались как удобный инструмент в теории и практике приближения функций. Однако довольно скоро область их применения начала быстро расширяться и обнаружилось, что существует очень много сплайнов самых разных типов. Сплайны стали активно использоваться в численных методах, в системах автоматического проектирования и автоматизации научных исследований, во многих других областях человеческой деятельности и, конечно, в компьютерной графике.

Сам термин "сплайн" происходит от английского spline. Именно так называется гибкая полоска стали, при помощи которой чертежники проводили через заданные точки плавные кривые. В былые времена подобный способ построения плавных обводов различных тел, таких, как, например, корпус корабля, кузов автомобиля, а потом фюзеляж или крыло самолета, был довольно широко распространен в практике машиностроения. В результате форма тела задавалась при помощи набора очень точно изготовленных сечений - плазов. Появление компьютеров позволило перейти от этого, плазово-шаблонного, метода к более эффективному способу задания поверхности обтекаемого тела. В основе этого подхода к описанию поверхностей лежит использование сравнительно несложных формул, позволяющих восстанавливать облик изделия с необходимой точностью. Ясно, что для большинства тел, встречающихся на практике, вряд ли возможно отыскание простых универсальных формул, которые описывали бы соответствующую поверхность глобально, то есть, как принято говорить, в целом. Это означает, что при решении задачи построения достаточно произвольной поверхности обойтись небольшим количеством формул, как правило, не удается. Вместе с тем аналитическое описание (описание посредством формул) внешних обводов изделия, то есть задание в трехмерном пространстве двумерной поверхности, должно быть достаточно экономным. Это особенно важно, когда речь идет об обработке изделий на станках с числовым программным управлением. Обычно поступают следующим образом: задают координаты сравнительно небольшого числа опорных точек, лежащих на искомой поверхности, и через эти точки проводят плавные поверхности. Именно так поступает конструктор при проектировании кузова автомобиля (ясно, что на этой стадии процесс проектирования сложного объекта содержит явную неформальную составляющую). На следующем шаге конструктор должен получить аналитическое представление для придуманных кривых или поверхностей. Вот для таких задач и используются сплайны.

Средства компьютерной графики, особенно визуализация, существенно помогают при проектировании, показывая конструктору, что может получиться в результате, и давая ему многовариантную возможность сравнить это с тем, что сложилось у него в голове.

Мы не ставим перед собой и читателем задачи рассказать обо всех сплайнах, в частности потому, что это отдельная большая тема, требующая и большего внимания, и большего объема. Во вводном курсе нам кажется более уместным показать в сравнении некоторые из преимуществ использования сплайнов в задачах геометрического моделирования при проектировании кривых и поверхностей. Такое представление полезно начинающему пользователю для его ориентации в стремительно расширяющемся мире сплайнов. При этом мы ограничимся лишь сплайнами, в построении которых используются кубические (в случае одномерных сплайнов - сплайновых кривых) и бикубические (в случае двумерных сплайнов - сплайновых поверхностей) многочлены. В компьютерной графике подобные сплайны применяются наиболее часто.

Достаточно типичной является следующая задача: по заданному массиву точек на плоскости (2D) или в пространстве (3D) построить кривую либо проходящую через все эти точки (задача интерполяции), либо проходящую вблизи от этих точек (задача сглаживания).

Совершенно естественно возникают вопросы: 1) в каком классе кривых искать решение поставленной задачи? и 2) как искать?

Сплайн-функции

А. Случай одной переменной

Обратимся для определенности к задаче интерполяции и начнем рассмотрение с обсуждения правил выбора класса кривых.

Ясно, что допустимый класс кривых должен быть таким, чтобы решение задачи было единственным (это обстоятельство сильно помогает в преодолении многих трудностей поиска). Кроме того, желательно, чтобы построенная кривая изменялась плавно.

Пусть на плоскости задан набор точек (x_i, y_i), i = 0, 1, ..., m, таких, что $x_0 < x_1 < ... < x_{m-1} < x_m$.

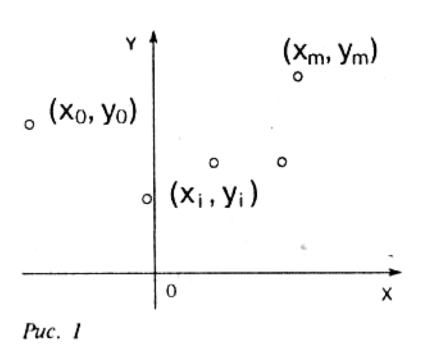


Рис. 1

То обстоятельство, что точки заданного набора занумерованы в порядке возрастания их абсцисс, позволяет искать кривую в классе графиков функций. Мы сможем описать основные проблемы сглаживания этого дискретного набора, ограничившись случаем многочленов.

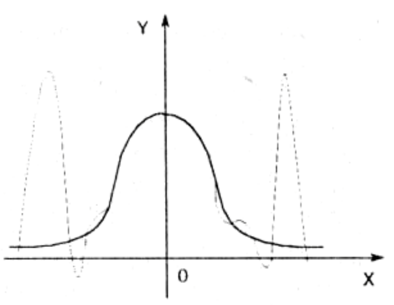


Рис. 2

Как известно из курса математического анализа, существует интерполяционный многочлен Лагранжа

$$L_m(x) = \sum_{i=0}^{m} y_i \frac{\omega_m(x)}{(x-x_i)\omega_m(x_i)},$$

где $\omega_m(x) = \prod_{j=0}^{m}(x-x_j)$,

график которого проходит через все заданные точки (x_i, y_i), i = 0, 1,..., m ,

Это обстоятельство и простота описания (заметим, что многочлен однозначно определяется набором своих коэффициентов; в данном случае их число совпадает с количеством точек в заданном наборе) являются несомненными достоинствами построенного интерполяционного многочлена (разумеется, есть и другие).

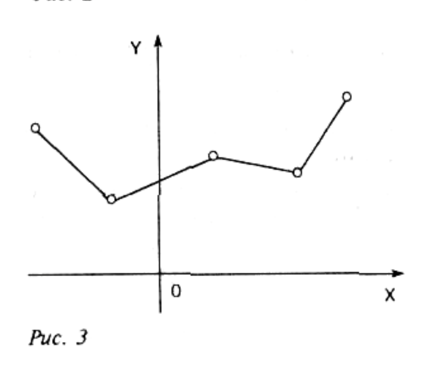


Рис. 3

Однако нам полезно остановиться и на некоторых недостатках предложенного подхода.

1. Степень многочлена Лагранжа на единицу меньше числа заданных точек. Поэтому, чем больше точек задано, тем выше степень такого многочлена. И хотя график интерполяционного многочлена Лагранжа всегда будет проходить через все точки массива, его уклонение (от ожидаемого) может оказаться довольно значительным (рис. 2).

2. Изменение одной точки (ситуация, довольно часто встречаемая на практике) требует полного пересчета коэффициентов интерполяционного многочлена и к тому же может существенно повлиять на вид задаваемой им кривой.

Приближающую кривую можно построить и совсем просто: если последовательно соединить точки заданного набора прямолинейными отрезками, то в результате получится ломаная (рис. 3). При такой, кусочно-линейной, интерполяции требуется найти всего 2m чисел (каждый прямолинейный отрезок определяется ровно двумя коэффициентами), но, к сожалению, построенная таким образом аппроксимирующая кусочно-линейная функция не

обладает нужной гладкостью: уже первая производная этой функции терпит разрывы в узлах интерполяции.

Рассмотрев эти две крайние ситуации, попробуем найти класс функций, которые в основном сохранили бы перечисленные выше достоинства обоих подходов и одновременно были бы в известной степени свободны от их недостатков.

Для этого поступим так: будем использовать многочлены (как и в первом случае) и строить их последовательно, звено за звеном (как во втором случае). В результате получится так называемый полиномиальный многозвенник. При подобном подходе важно правильно выбрать степени привлекаемых многочленов, а для плавного изменения результирующей кривой необходимо еще тщательно подобрать коэффициенты многочленов (из условий гладкого сопряжения соседних звеньев).

То, что получится в результате описанных усилий, называют сплайн-функциями или просто сплайнами.

Для того, чтобы понять, какое отношение имеют сплайн-функции к чертежным сплайнам, возьмем гибкую стальную линейку, поставим ее на ребро и, закрепив один из концов в заданной точке, поместим ее между опорами, которые располагаются в плоскости Oxy в точках (x_i, y_i), $i = 0,1,...m$, где $x_0 < x_1 < ...< x_{m-1} < x_m$ (Рис.4).

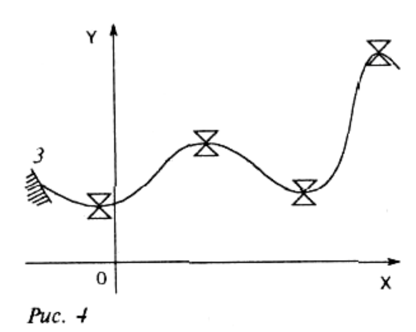


Рис. 4

Интересно отметить, что функция

$y = S(x)$,

описывающая профиль линейки, обладает следующими интересными свойствами:

- с довольно большой точностью часть графика этой функции, заключенную между любыми двумя соседними опорами, можно считать многочленом третьей степени;
- на всем промежутке [x0, xm] функция $y = S(x)$ дважды непрерывно дифференцируема.

Построенная функция S(x) относится к так называемым интерполяционным кубическим сплайнам. Этот класс в полной мере удовлетворяет высказанным выше требованиям и обладает еще целым рядом замечательных свойств.

Перейдем, однако, к точным формулировкам.

Интерполяционным кубическим сплайном называется функция S(x), обладающая следующими свойствами:

1) график этой функции проходит через каждую точку заданного массива,

$S(x_i) = y_i$, $i = 0, 1,..., m$;

2) на каждом из отрезков

$[x_i, x_{i+1}]$, $1 = 0, 1, ..., m - 1$,

функция является многочленом третьей степени,

$$S(x) = \sum_{j=0}^{3} a_j^i (x - x_i)^j;$$

3) на всем отрезке задания $[x_0,x_m]$функция S(x) имеет непрерывную вторую производную.

Так как на каждом из отрезков $[x_i, x_{i+1}]$ сплайн S(x) определяется четырьмя коэффициентами, то для его полного построения на всем отрезке задания необходимо найти 4m чисел.

Третье условие будет выполнено, если потребовать непрерывности сплайна во всех внутренних узлах x_i; $i = 1, ..., m-1$ (это дает m-1 условий на коэффициенты), а также его первой (m-1 условий) и второй (еще m-1 условий) производных в этих узлах. Вместе с первым условием получаем

$m-1 + m-1 + m-1 + m+1 = 4m - 2$

равенства. Недостающие два условия для полного определения коэффициентов можно получить, задав, например, значения первых производных на концах отрезка [x_0,x_m] (граничные условия):

$S'(x_0) = l_0, S'(x_m) = l_m$.

Существуют граничные условия и других типов.

Б. Случай двух переменных

Более сложная задача построения по заданному набору точек в трехмерном пространстве интерполяционной функции двух переменных решается похожим образом.

Расскажем прежде всего, что такое интерполяционный бикубический сплайн.

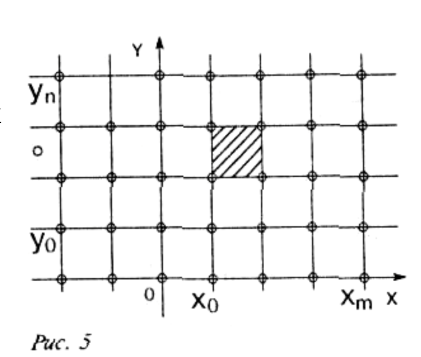


Рис. 5

Пусть на плоскости задан набор из $(m+1)(n+1)$ точек (рис. 5)

$(x_i, y_j), i = 0,1,\ldots,m; j = 0,1,\ldots,n,$

где $x_0 < x_1 <\ldots< x_{m-1} < x_m$, $y_0 < y_1 < \ldots < y_{n-1} < y_n$.

Добавим к каждой паре (x_i, y_j) третью координату Z_{ij} - (x_i, y_j, z_{ij}).

Тем самым мы получаем массив

(x_j, y_j, z_{ij}), i = 0, 1,..., m; j = 0, l,...,n.

Прежде чем строить поверхность, проходящую через все точки заданного массива, определим функцию, графиком которой будет эта поверхность.

Интерполяционным бикубическим сплайном называется функция двух переменных S(x, y), обладающая следующими свойствами:

1) график этой функции проходит через каждую точку заданного массива,

$S(x_i,y_j) = z_{ij}$, i = 0, l,...,m; j = 0,1,.., n ;

2) на каждом частичном прямоугольнике

$\lfloor x_i, x_{j+1} \rfloor \times \lfloor y_j, y_{j+1} \rfloor, i = 0,1,\ldots,m-1, j = 0,1,\ldots,n-1,$

функция представляет собой многочлен третьей степени по каждой из переменных,

$S(x,y) = \sum_{l,k=0}^{3} a_{lk}^{ij} (x - x_i)^l (y - y_j)^k;$

3) на всем прямоугольнике задания $[x_0, x_m] \times [y_0, y_n]$ функция S(x, y) имеет по каждой переменной непрерывную вторую производную.

Для того чтобы построить по заданному массиву {(x_i, y_j, z_{ij})} интерполяционный бикубический сплайн, достаточно определить все 16mn коэффициентов. Как и в одномерном случае, отыскание коэффициентов сплайн-функции сводится к построению решения системы

линейных уравнений, связывающих искомые коэффициенты a_{lk}^{ij}.

Последняя возникает из первого и третьего условий после добавления к ним недостающих соотношений путем задания значений производной искомой функции в граничных узлах прямоугольника $[x_0, x_m] \times [y_0, y_n]$ (или иных соображений).

Подведем некоторые итоги.

Достоинства предложенного способа несомненны: для решения линейных систем, возникающих в ходе построения сплайн-функций, существует много эффективных методов, к тому же эти системы достаточно просты; графики построенных сплайн-функций проходят через все заданные точки, полностью сохраняя первоначально заданную информацию.

Вместе с тем изменение лишь одной точки (случай на практике довольно типичный) при описанном подходе заставляет пересчитывать заново, как правило, все коэффициенты.

К тому же во многих задачах исходный набор точек задается приближенно и, значит, требование неукоснительного прохождения графика искомой функции через каждую точку этого набора оказывается излишним.

От этих недостатков свободны некоторые из методов сглаживания, к описанию которых мы и переходим. Но прежде всего мы значительно расширим классы, в которых будет вестись поиск соответствующих кривых и поверхностей. Более точно, мы откажемся от требования однозначного проектирования искомой кривой на координатную ось, а поверхности - на координатную плоскость. Такой подход позволяет ослабить и требования к задаваемому массиву.

Сказанное требует небольшого геометрического введения.

Начнем, как и прежде, с кривых.

Сплайновые кривые

Нам будет удобно пользоваться параметрическими уравнениями кривой. Напомним необходимые понятия.

Параметрически заданной кривой называется множество γ точек M(x, y, z), координаты x, y, z которых определяются соотношениями

$$x = x(t), y = y(t), z = z(t),$$
$$a \le t \le b, \qquad (1)$$

где x(t), y(t), z(t) - функции, непрерывные на отрезке [a, b] (рис. 6).

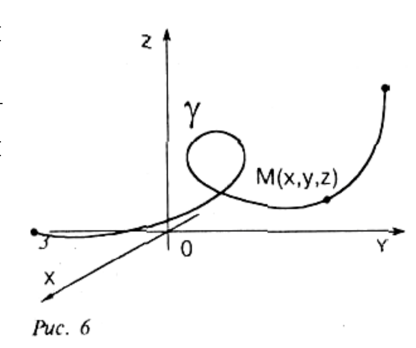


Рис. 6

Соотношения (1) называются параметрическими уравнениями кривой γ.

Без ограничения общности можно считать, что a = 0 и b = 1; этого всегда можно добиться при помощи замены вида

$$u = \frac{t-a}{b-a}.$$

Полезна векторная форма записи параметрических уравнений

$$r = r(t), 0 \le t \le 1,$$

где $r(t) = (x(t), y(t), z(t))$.

Параметр t задает ориентацию параметризованной кривой γ (порядок прохождения точек при монотонном возрастании параметра).

Рис. 7

Кривая γ называется регулярной кривой, если r'(t) ≠ 0 в каждой ее точке. Это означает, что в каждой точке кривой существует касательная к ней и эта касательная меняется непрерывно вслед за перемещающейся вдоль кривой ее текущей точки (рис. 7). Единичный вектор касательной к кривой γ равен

$$T(t) = \frac{r'(t)}{|r'(t)|}.$$

Если дополнительно потребовать, чтобы задающая кривую векторная функция имела вторую производную, то будет определен вектор кривизны кривой

$$K(t) = \frac{[r'(t) \times r''(t)] \times r'(t)}{|r'(t)|^4},$$

модуль которого характеризует степень ее отклонения от прямой (рис. 8). В частности, если γ - отрезок прямой, то K = 0.

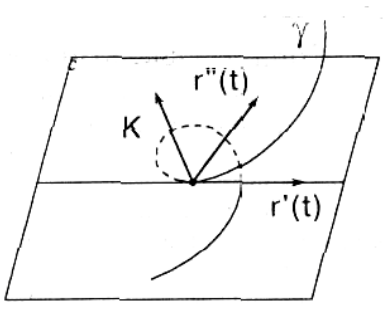


Рис. 8

Замечание

При дальнейшем изложении мы имеем в виду расположение рассматриваемых объектов в трехмерном пространстве. Практически все сказанное будет верно и для плоского случая (более общего, чем рассмотренный выше). Дело в том, что параметрическое описание плоской кривой не накладывает никаких ограничений на ее расположение относительно координатных осей: кривая не обязана однозначно проектироваться на координатную ось, как это имеет место в случае ее явного задания y = y(x). В частности, кривая может быть замкнутой, самопересекающейся и так далее. Все последующие построения законны и в этих сложных случаях.

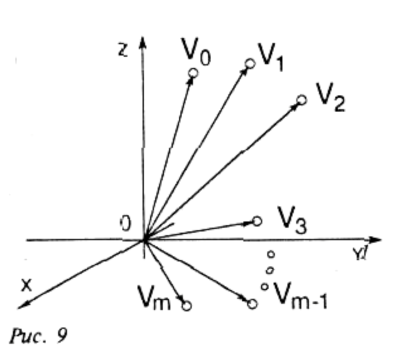


Рис. 9

Рассмотрим некоторые подходы к построению сглаживающей кривой. Пусть на плоскости или в пространстве задан упорядоченный набор точек, определяемых векторами V_0, V_1, ..., V_m (рис. 9). Ломаная $V_0V_1...V_m$ называется контрольной ломаной, порожденной массивом $V = \{V_0, V_1,..., V_m\}$ (Рис. 10).

Кривой Безье, определяемой массивом V, называется кривая, определяемая векторным уравнением

$$r(t) = \sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} V_i, 0 \le t \le 1, \qquad (2)$$

где $C_m^j = \dfrac{m!}{i!(m-i)!}$ –

коэффициенты в разложении бинома Ньютона (число сочетаний из m элементов по i).

Рис. 10

Кривая Безье обладает замечательными свойствами:

- она является гладкой;
- начинается в точке V_0 и заканчивается в точке V_m, касаясь при этом отрезков V_0V_1 и $V_{m-1}V_m$ контрольной ломаной;
- функциональные коэффициенты $C_m^i t^i (1-t)^{m-i}$ при вершинах Vi, i = 0, 1, ..., m, суть универсальные многочлены (многочлены Бернштейна); они неотрицательны, и их сумма равна единице:

$$\sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} = (t + (1-t))^m = 1$$

Поэтому кривая Безье целиком лежит в выпуклой оболочке, порождаемой массивом (рис. 11).

При m = 3 получаем (элементарную) кубическую кривую Безье, определяемую четверкой точек V_0, V_1, V_2, V_3 и описываемую векторным параметрическим уравнением

$$r(t) = (((1-t)V_0 + 3tV_1)(1-t) + 3t^2V_2)(1-t) + t^3V_3,$$

$$0 \le t \le 1,$$

или, в матричной записи, r(t) = VMT , 0 ≤ t ≤ 1,

$$\text{где } r(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}, V = \begin{pmatrix} V_0 & V_1 & V_2 & V_3 \end{pmatrix} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{pmatrix},$$

$$M = \begin{pmatrix} 1 & -3 & 3 & 1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}, T = \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Матрица M называется базисной матрицей кубической кривой Безье.

Порядок точек в заданном наборе существенно влияет на вид элементарной кривой Безье. На рис. 12 построены элементарные кубические кривые Безье, порожденные наборами четверок точек, которые различаются только нумерацией. Нетрудно заметить, что, находясь в одной и той же выпуклой оболочке и пытаясь повторить контрольную ломаную в гладком варианте, эти кривые заметно разнятся.

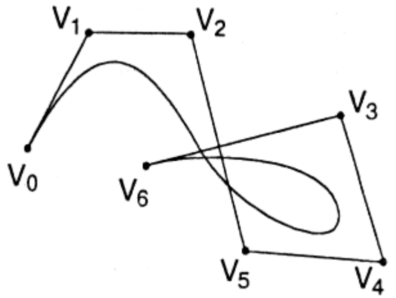


Рис. 11

Наряду с отмеченными достоинствами кривые Безье обладают и определенными недостатками.

Основных недостатков у элементарных кривых Безье три:

1) степень функциональных коэффициентов напрямую связана с количеством точек в заданном наборе (на единицу меньше);

2) при добавлении хотя бы одной точки в заданный набор необходимо провести полный пересчет функциональных коэффициентов в параметрическом уравнении кривой;

3) изменение хотя бы одной точки приводит к заметному изменению вида всей кривой.

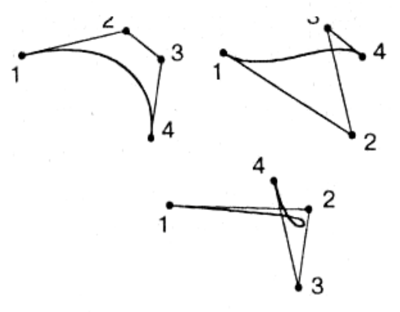


Рис. 12

В практических вычислениях часто оказывается удобным пользоваться кривыми, составленными из элементарных кривых Безье, как правило кубических.

Важное замечание

При построении кривой из определенным образом подобранных фрагментов важна не только регулярность самих этих фрагментов, но и выполнение некоторых условий гладкости в точках их состыковки. Только в этом случае составная кривая, получающаяся в результате проведенных построений, будет обладать достаточно хорошими геометрическими характеристиками. Однако при построении составных кривых часто приходится сталкиваться с ситуацией, когда каждый из регулярных фрагментов, участвующих в создании новой кривой, имеет свою собственную параметризацию. Чтобы учесть это обстоятельство, удобно использовать класс так называемых геометрически непрерывных кривых.

Составная кривая называется G^1 -(геометрически) непрерывной, если вдоль этой кривой единичный вектор ее касательной изменяется непрерывно, и G^2-(геометрически) непрерывной, если вдоль этой кривой изменяется непрерывно, кроме того, и вектор кривизны. Обратимся к рассмотрению составных кривых Безье. Составная кубическая кривая Безье представляет собой объединение элементарных кубических кривых Безье $\gamma_1,\dots,\gamma_m$, таких, что

$r_i(1) = r_{i+1}(0)$, $i = 0,\dots,m\text{-}l$,

где $r = r_i(t)$, $0 \le t \le 1$, - параметрическое уравнение кривой γ_i.

Чтобы составная кривая Безье, определяемая набором вершин

$V_0, V_1, \dots, V_{m-1}$,

1) была G^1-непрерывной кривой, необходимо, чтобы каждые три точки

$V_{3i-1}, V_{3i}, V_{3i+1}$

этого набора лежали на одной прямой;

2) была замкнутой G^1-непрерывной кривой, необходимо, кроме того, чтобы совпадали первая и последняя точки,

$V_0 = V_m$,

и три точки

$V_{m-1}, V_m = V_0, V_1$

лежали на одной прямой;

3) была G^2-непрерывной кривой, необходимо, чтобы каждые пять точек

$V_{3i-2}, V_{3i-1}, V_{3i}, V_{3i+1}, V_{3i+2} (i \ge 1)$

заданного набора лежали в одной плоскости.

```
// File Bezier.cpp
#include <math.h>
double Bezier ( double p [], int i, double t )
{
   double s  = 1 - t;
   double t2 = t * t;
   double t3 = t2 * t ;
   return ((p[3*i]*s+3*t-p[ 3*i+1 ])*s+3*t2*p[3*i+2])*s+t3*p[3*i+3];
}
```

Попытаемся найти другой класс кривых, сохраняющих перечисленные достоинства кривых Безье и лишенных их недостатков.

Так как в векторном уравнении, задающем кривую Безье, векторные составляющие постоянны (это просто вершины массива), то мы уделим основное внимание выбору новых функциональных коэффициентов, стараясь (разумеется, по возможности) сохранить при этом замечательные свойства многочленов Бернштейна, ограничив наши рассмотрения кубическими многочленами.

По заданному набору точек

V_0, V_1, V_2, V_3

элементарная кубическая В-сплайновая кривая определяется при помощи векторного параметрического уравнения следующего вида:

$$r(t) = \frac{(1-t)^3}{6}V_0 + \frac{3t^3 - 6t^2 + 4}{6}V_1 + \frac{-3t^3 + 3t^2 + 3t + 1}{6}V_2 + \frac{t^3}{6}V_3,$$

$0 \le t \le 1,$

или, в матричной форме,

$r(t) = VMT, 0 \le t \le 1,$

где $r(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}, V = \begin{pmatrix} V_0 & V_1 & V_2 & V_3 \end{pmatrix} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{pmatrix},$

$$M = \begin{pmatrix} 1 & -3 & 3 & 1 \\ 4 & 0 & -6 & 3 \\ 1 & 2 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}, T = \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Матрица M называется базисной матрицей В-сплайновой кривой.

Функциональные коэффициенты в уравнении, определяющем элементарную В-сплайновую кубическую кривую, неотрицательны, в сумме составляют единицу, универсальны (не зависят от конкретного вида точек в заданной четверке).

Это означает, что рассматриваемый элементарный фрагмент лежит внутри выпуклой оболочки заданных вершин - четырехугольника (в плоском случае) или тетраэдра (в пространственном случае) (рис. 13).

Рис. 13

Составная кубическая В-сплайновая кривая, задаваемая параметрическим уравнением

$r = r(t), 0 \le t \le m-2,$

и определяемая набором точек

$V_0, V_1, \dots, V_{m-1}, V_m$ ($m \ge 3$),

представляет собой объединение m-2 элементарных кубических B-сплайновых кривых, $\gamma_3, \dots, \gamma_m$, описываемых уравнениями вида

$$r = r_i(t) = \begin{pmatrix} V_{i-1} & V_i & V_{i+1} & V_{i+2} \end{pmatrix} M \begin{pmatrix} 1 \\ t-i+1 \\ (t-i+1)^2 \\ (t-i+1)^3 \end{pmatrix},$$

$i-1 \le t \le i, i = 1, \dots, m-2.$

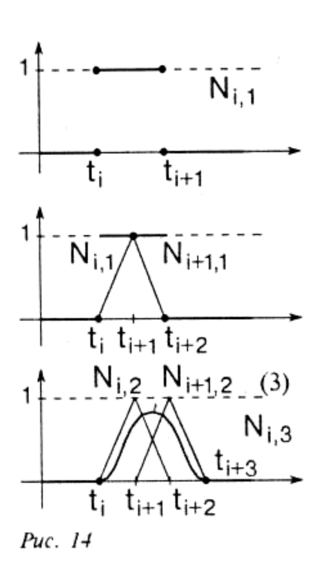


Рис. 14

Замечание

Область изменения параметра t и расположение на ней точек, соответствующих стыкованным узлам, могут быть совершенно произвольными. Наиболее простой является равномерная параметризация с равноотстоящими целочисленными узлами.

Указанная выше составная B-сплайновая кубическая кривая является C^2-гладкой кривой и лежит в объединении m-2 выпуклых оболочек, порожденных последовательными четверками точек заданного набора. Добавляя в исходный набор дополнительные точки, можно получать составную B-сплайновую кубическую кривую с разными свойствами.

Например, при добавлении к заданному набору двух точек

$V_{-1} = (V_0 - V_1) + V_0, V_{m+1} = (v_m - v_{m-1}) + v_m$

и соответствующем расширении отрезка изменения параметра до [0, m] получим составную B-сплайновую кубическую кривую, которая

будет начинаться в точке V_0, касаясь отрезка V_0V_1, и заканчиваться в точке V_m, касаясь отрезка $V_{m-1}V_m$.

А для того, чтобы по заданному массиву построить C^2-гладкую замкнутую B-сплайновую кривую, достаточно выбрать три дополнительные точки

$V_{m+1} = V_0, V_{m+2} = V_1, V_{m+3} = V_2$

и рассмотреть набор

$V_0, V_1, V_2, \dots, V_m, V_{m+1}, v_{m+2}, v_{m+3}$.

```
// File BSpline.cpp
#include <math.h>
double BSpline ( double p [], int i, double t )
{
   double s = 1.0 - t;
   double t2 = t * t;
   double t3 = t2 * t;
   return (s*s*s*p[i] + (3*t3 - 6*t2 + 4) * p[i+1] + (-3*t3 + 3*t2 + 3*t + 1) * p[i+2]
+ t3*p[i+3]) / 6.0;
}
```

Перейдем к случаю, когда узлы расположены на отрезке изменения параметра t неравномерно.

Заменим в векторном уравнении (2) многочлены Бернштейна на B-сплайны (базовые (base) сплайны), введя новые функциональные коэффициенты при помощи рекуррентных формул.

Пусть $0 = t_0 < t_1 < \dots < t_{m-1}, < t_m = 1$ - разбиение отрезка 0,1,

Положим

$N_{i,1}(t) = 1, t \in [t_i, t_{i+1}],$

$N_{i,1}(t) = 0, t \in [t_i, t_{i+1}]$

и далее (рис. 14)

$$N_{i,q}(t) = \frac{t - t_i}{t_{i+q-1} - t_i} N_{i,q-1}(t) + \frac{t_{i+q} - t}{t_{i+q} - t_{i+1}} N_{i+1,q-1}(t).$$

Заметим, что с увеличением индекса q степень многочленов, определяющих вводимые функции $N_{i,q}(t)$, растет: для функций на отрезке $[t_i, t_{i+q}]$ она равна q-1.

Отметим еще некоторые очевидные свойства этих функций:

- $N_{i,q}(t) > 0$ на интервале (t_i, t_{i+q});

- $N_{i,q}(t) = 0$ вне интервала (t_i, t_{i+q});
- на всей области задания функция $N_{i,q}(t), q \geq 3$, имеет непрерывные производные до порядка q-2 включительно.

Кроме того, для введенных функций сохраняется равенство

$\sum_{i=0}^{m} N_{i,q}(t) = 1.$

Это означает, что кривая, заданная векторным уравнением

$r(t) = \sum_{i=0}^{m} N_{i,q}(t) V_i,$

всегда принадлежит выпуклой оболочке вершин заданного массива.

Замечание

На самом деле кривая лежит в объединении выпуклых оболочек, порожденных последовательными наборами из q + 1 точек заданного массива. Построенная кривая обладает важным локальным свойством: изменение одной вершины в массиве (или добавление новой вершины к имеющимся) уже не ведет, как прежде, к полному изменению всей кривой.

В силу третьего свойства сохраняется достаточная гладкость кривой: если взять q ≥ 4, то все функциональные коэффициенты будут иметь непрерывные вторые производные. Для практических задач большей гладкости, как правило, не требуется. Поэтому обычно ограничиваются рассмотрением случая, когда q = 4.

Замечание

Для построения кубического B-сплайна N_i,4(t) требуется 5 узлов разбиения

l_i, l_{i+1}, l_{i+2}, l_{i+3}, l_{i+4} отрезка [0, 1]. Поэтому если узлов не хватает, то их набор определенным образом расширяют, например полагая.

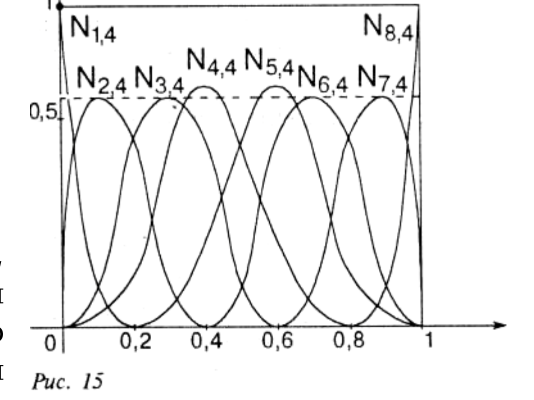


Рис. 15

Дополнительно введенные отрезки имеют нулевую длину, и первоначальные первый $t_0 = 0$ и последний $t_m = 1$ узлы становятся кратными. На рис. 15 показан полный набор кубических B-сплайнов, построенных на расширенном множестве узлов

$t_{-3} = t_{-2} = t_{-1} = t_0 = 0;$

$t_1 = 0{,}2;\ t_2 = 0{,}4;\ t_3 = 0{,}6;\ t_4 = 0{,}8;$

$t_s = t_6 = t_7 = t_8 = 1.$

Замечание

Выбор узлов параметризации может быть совершенно произвольным. Однако часто удобной оказывается параметризация, в которой промежуток изменения параметра t и узлы t_i определяются длинами соответствующих хорд:

$$t_0 = 0,$$

$$t_1 = |V_2 V_0|,$$

$$t_i = t_{i-1} + |V_{i+1} V_{i+2}|, i = 2,\dots, m-3,$$

$$t_{m-2} = t_{m-3} + |V_m V_{m-2}|.$$

Обратимся к следующей достаточно типичной ситуации: по заданному набору точек мы построили В-сплайновую кривую, вывели полученный результат на экран и, внимательно изучив то, что предстало перед глазами, ощутили необходимость подправить кривую в одном или нескольких местах, не изменяя исходного набора точек. Наиболее подходящим инструментом для подобной процедуры являются числовые или функциональные параметры, заранее введенные в уравнения кривых.

Такую возможность предоставляют некоторые обобщения кубических В-сплайнов, а именно рациональные кубические В-сплайны и бета-сплайны, к описанию которых мы и обратимся.

Рациональные кубические В-сплайны

По заданному набору

V_0, V_1, V_2, V_3

рациональная кубическая В-сплайновая кривая определяется уравнением следующего вида:

$$r(t) = \frac{\sum_{i=0}^{3} w_i n_i(t) V_i}{\sum_{i=0}^{3} w_i n_i(t)}, 0 \le t \le 1,$$

где

$$n_0(t) = \frac{(1-t)^3}{6}, n_1(t) = \frac{3t^3 - 6t^2 + 4}{6},$$

$$n_2(t) = \frac{-3t^3 + 3t^2 + 3t + 1}{6}, n_3(t) = \frac{t^3}{6},$$

а величины W_i, называемые весами (или параметрами формы), -неотрицательные числа, сумма которых положительна.

Замечания:

1. В случае, если все веса равны между собой, приведенное уравнение описывает элементарную кубическую В-сплайновую кривую.

2. Построение составной рациональной В-сплайновой кубической кривой проводится по той же схеме, что и в полиномиальном случае.

3. В последнее время значительный интерес пользователей вызывает класс сплайнов, известный под названием NURBS - nonuniform rational B-splines - рациональных В-сплайнов, задаваемых на неравномерной сетке.

Бета-сплайны

Применение составных бета-сплайновых кривых основывается на важном свойстве геометрической непрерывности.

Как отмечалось выше, в построении составной регулярной кривой важную роль играют условия сопряжения в точках контакта слагающих ее отрезков регулярных кривых.

Пусть γ_1 и γ_2 - регулярные кривые, заданные параметрическими уравнениями

$$r = r_1(t), 0 \le t \le 1; r = r_2(t), 0 \le t \le 1,$$

соответственно и имеющие общую точку

$$r_1(1) = r_2(t). \quad (4)$$

Для того, чтобы кривая γ, составленная из кривых γ_1 и γ_2, была регулярной, потребуем совпадения в общей точке единичных касательных векторов

$$\frac{r'(1)}{|r'(1)|} = \frac{r'(0)}{|r'(0)|} \quad (5)$$

и векторов кривизны

$$\frac{[r_1'(1) \times r_1''(1)] \times r_1'(1)}{\left|r_1'(1)\right|^4} = \frac{[r_2'(1) \times r_2''(1)] \times r_2'(1)}{\left|r_2'(1)\right|^4}, \quad (6)$$

сопрягаемых кривых γ_1 и γ_2.

Нетрудно проверить, что если радиусы-векторы кривых γ_1 и γ_2 связаны условиями геометрической непрерывности

$$r_2(0) = r_1(1),$$
$$r_2'(0) = \beta_1 r_1'(1), \quad (7)$$
$$r_2''(0) = \beta_1^2 r_1''(1) + \beta_2 r_1'(1).$$

где $\beta_i > 0$, $\beta_2 >= 0$ - числовые параметры, то каждое из условий (4)-(6) будет выполнено.

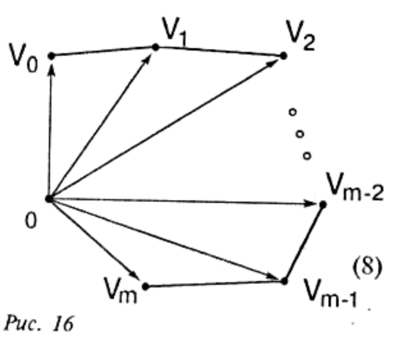


Рис. 16

Рассмотрим набор из m+1 точек V_0, V_1, ..., V_{m-1}, V_m, заданных своими радиусами-векторами (рис. 16). Будем искать сглаживающую составную регулярную кривую γ при помощи частичных кривых γ_i, описываемых уравнениями вида

$$r_i(t) = \sum_{j=-2}^{1} b_j(t) V_{ij}, 0 \le t \le 1, \quad (8)$$

где $\begin{cases} b_j(t) = \sum_{k=0}^{3} c_{kj}(\beta_1, \beta_2) t^k, \\ j = -2, -1, 0, 1 \end{cases} \quad (9)$

не зависящие от i весовые функциональные коэффициенты. Для того, чтобы найти эти весовые коэффициенты, потребуем, чтобы векторы $r_i(t)$ и $r_{i+1}(t)$ в точке сопряжения удовлетворяли условиям геометрической непрерывности (7). С учетом формул (8) эти условия можно записать так:

$$\sum_{j=-2}^{1} b_j(0) V_{i+1+j} = \sum_{j=-2}^{1} b_j(1) V_{i+j},$$
$$\sum_{j=-2}^{1} b_j'(0) V_{i+1+j} = \beta_1 \sum_{j=-2}^{1} b_j'(1) V_{i+j}, \quad (10)$$
$$\sum_{j=-2}^{1} b_j''(0) V_{i+1+j} = \beta_1^2 \sum_{j=-2}^{1} b_j''(1) V_{i+j} + \beta_2 \sum_{j=-2}^{1} b_j'(1) V_{i+j}.$$

Полученные соотношения позволяют найти все функциональные коэффициенты $b_j(t)$, j = -2, - 1, 0, 1.

Расписав, например, первое из равенств (10) подробнее:

$$b_{-2}(0)V_{i-1} + b_{-1}(0)V_i + b_0(0)V_{i+1} + b_1(0)V_{i+2} =$$
$$= b_{-2}(1)V_{i-2} + b_{-1}(1)V_{i-1} + b_0(l)V_i + b_1(1)V_{i+1} -$$

и приравняв коэффициенты при одинаковых векторах, получим:

$$0 = b_{-2}(1), b_{-2}(0) = b_{-1}(1), b_{-1}(0) = b_0(1), b_0(0) = b_1(l), b_1(0) = 0.$$

Подобным же образом из двух других векторных равенств (10) получаются соотношения, связывающие значения в точках 0 и 1 первых и вторых производных функциональных коэффициентов.

Привлекая формулы (9), получаем в итоге линейную систему для искомых чисел c_{kj}, определитель которой

$$\delta = 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + \beta_2 + 2 > 0.$$

Разрешая линейную алгебраическую систему, найдем величины c_{kj} и затем подставим полученные выражения в формулы (9).

Выражения для функциональных коэффициентов

$$b_{-2}(t) = \frac{2\beta_1^3}{\delta}(1-t)^3,$$

$$b_{-1}(t) = \frac{1}{\delta}\left[2\beta_1^3 t(t^2 - 3t + 3) + 2\beta_1^2(t^3 - 3t^2 + 2) + \right],$$

$$\left[2\beta_1(t^3 - 3t + 2) + \beta_2(2t^3 - 3t^2 + 1)\right]$$

$$b_0(t) = \frac{1}{\delta}\left[2\beta_1^2 t^2(-t + 3) + 2\beta_1 t(-t^2 + 3) + \beta_2 t^2(-2t + 3) + 2(-t^3 + 1)\right],$$

$$b_1(t) = \frac{2t^3}{\delta}$$

годятся для всей конструкции. Подставляя их в формулу (8), получаем значения векторных функций

$r_2(t), \ldots, r_{m-1}(t)$.

Рис. 17

Заметим, что кривая, определяемая векторной функцией ri(t) и, значит, вершинами V_{i-1}, V_i, V_{i+1}, V_{i+2}, лежит в их выпуклой оболочке (рис. 17).

Запишем уравнение элементарной бета-сплайновой кривой, порожденной набором точек V_{i-1}, Vi, V_{i+1}, V_{i+2}, в матричном виде. Имеем:

$$r(t) = VMT, 0 \le t \le 1,$$

Где $r(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}, T = \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix},$

$$V = \begin{pmatrix} V_{i-1} & V_i & V_{i+1} & V_{i+2} \end{pmatrix} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{pmatrix},$$

$$M = \frac{1}{\delta}\begin{pmatrix} 2\alpha & -6\alpha & 6\alpha & -2\alpha \\ 4(\beta_1^2 + \beta) + \beta_2 & 6(\alpha - \beta_1) & -3(2\alpha + \mu) & 2(\alpha + \upsilon) \\ 2 & 6\beta_1 & 3\mu & -2(\upsilon + 1) \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

Здесь

$$\alpha = \beta_1^3, \mu = 2\beta_1^2 + \beta_2, \nu = \beta_1^2 + \beta_1 + \beta_2.$$

Матрица M называется базисной матрицей бета-сплайновой кривой.

Замечания: 1. *Числовые параметры B_1 и B_2 называются параметрами формы бета-сплайновой кривой, причем первый из них называют параметром скоса, а второй - параметром напряжения.*

2. При $B_1 = 1$, $B_2 = 0$ получается кубическая B-сплайновая кривая.

Подбором дополнительных вершин можно влиять на поведение составной бета-сплайновой кривой вблизи ее концов. Например, для того, чтобы составная кривая γ проходила через вершины V_0 и V_m, касаясь отрезков V_0V_1 и $V_{m-1}V_m$ контрольной ломаной (рис. 18), следует добавить к полученному набору векторных функций еще 4:

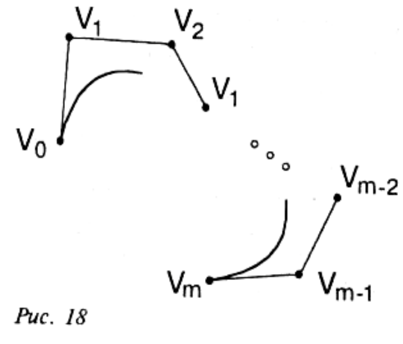


Рис. 18

$$r_0(t) = (1 - \frac{2t^3}{\delta})V_0 + \frac{2t^3}{\delta}V_1,$$

$$r_1(t) = [b_{-2}(t) + b_{-1}(t)]V_0 + b_0(t)V_1 + b_1(t)V_2,$$

$$r_m(t) = b_{-2}(t)V_{m-2} + b_{-1}(t)]V_{m-1} + [b_0(t) + b_1(t)]V_m,$$

$$r_{m+1}(t) = \frac{2\beta_1^3}{\delta})(1-t)^3V_{m-1} + [1 - \frac{2\beta_1^3}{\delta}(1-t)^3].$$

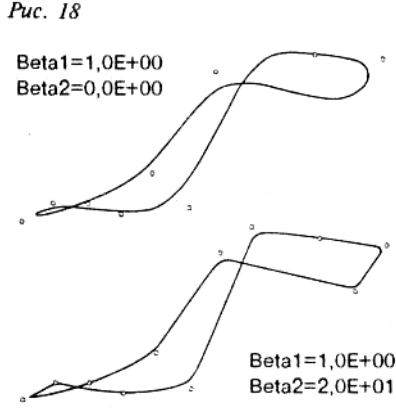
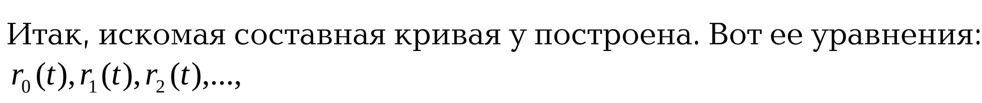


Рис. 19

Итак, искомая составная кривая y построена. Вот ее уравнения:



$r_0(t), r_1(t), r_2(t), \ldots,$

$r_{m-1}(t), r_m(t), r_{m+1}(t),$

$0 \le t \le 1.$

На рис. 19 показано, что изменение параметров β_1 и β_2 влечет изменение формы результирующей кривой.

```
// Beta.cpp
#include <math.h>
double BetaSpline (double beta1, double beta2, double p [],
int i, double t) {
double s = 1.0-t;
double t2 = t*t;
double t3 = t2*t;
double b12 = beta1*beta1;
double b13 = b12*beta1;
double delta = 20*b13+4.0*b12+4.0*beta1+beta2+2.0;
double d = 1.0 / delta;
double b0 = 2*b13*d*s8s*s;
double b3 = 2*t3*d;
double b1 = d*(2*b13*t*(t2-38t+3)+2*b12-(t3-3*t2+2) + 2*beta1*(t3-3*t+2)+beta2*(2*t3-3*t2+1));
double b2 = d*(2*b12*t2*(-t+3)+2*betal*t*(-t2+3)+ beta2*t2*(-2*t+3)+2*(-t3+1});
return b0*p [i] + b1*p [i+1] + b2*p [i+2] + b3*p [i+3]; }
```

Перейдем теперь к двумерному случаю - сплайновым поверхностям.

Сплайновые поверхности

Напомним некоторые понятия.

Регулярной поверхностью называется множество точек M(x, y, z) пространства, координаты x, y, z которых определяются из соотношений

$x = x(u,v)$, $y = y(u,v)$, $z = z(u,v)$, $(u,v) \in D$ (11)

где x(u,v), y(u, v), z(u,v) - гладкие функции своих аргументов, причем выполнено соотношение

$$rang\begin{pmatrix} x_u(u,v) & y_u(u,v) & z_u(u,v) \\ x_v(u,v) & y_v(u,v) & z_v(u,v) \end{pmatrix} = 2;$$

D - некоторая область на плоскости параметров u и v. Последнее равенство означает, что в каждой точке регулярной поверхности существует касательная плоскость и эта плоскость при непрерывном перемещении по поверхности текущей точки изменяется непрерывно (рис. 20).

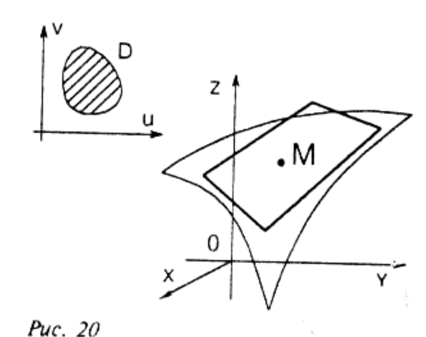


Рис. 20

Уравнения (11) называются параметрическими уравнениями поверхности. Их часто записывают также в векторной форме:

r = r(u, v), (u, v) ∈ D,

где

r(u, v) = (x(u, v), y(u,v),

Будем считать для простоты, что область на плоскости параметров представляет собой стандартный единичный квадрат (рис. 21).

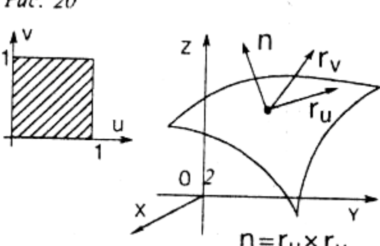
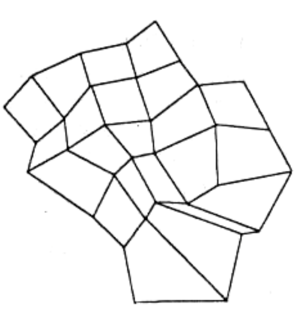


Рис. 21

Ограничим наши рассмотрения наборами точек вида

V_{ij}, i = 0, 1, ..., m; j = 0, 1, ... n.

Соединяя соответствующие вершины прямолинейными отрезками, получаем контрольный многогранник (точнее, контрольный, или опорный, граф) заданного массива V (рис. 22).


Рис. 22

Сглаживающая поверхность строится относительно просто, в виде так называемого тензорного произведения.

Так принято называть поверхности, описываемые параметрическими уравнениями вида

$$r(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_i(u) b_j(v) V_{ij},$$

где $\alpha \le u \le \beta, \gamma \le v \le \delta$.

То обстоятельство, что приведенное выше уравнение можно записать в следующей форме:

$$r(u,v) = \sum_{i=0}^{m} a_i(u) r_i(v),$$

где $r_i(v) = \sum_{j=0}^{n} b_j(v) V_{ij}, i = 0,...,m,$

позволяет переносить на двумерный случай многие свойства, результаты и наблюдения, полученные при исследовании кривых. Если при проводимом обобщении не сильно отклоняться от рассмотренных выше классов кривых, то так построенные поверхности будут "наследовать" многие свойства одноименных кривых. В этом бесспорное преимущество задания поверхности в виде тензорного произведения.

Замечание

При повышении размерности задачи неизбежно возникает значительное число новых проблем. Предложенные ограничения на структуру заданного набора точек (естественно обобщающую структуру плоского сеточного прямоугольника) и выбор в качестве рабочих наиболее простых классов поверхностей дают определенную возможность удержать это число в рамках, разумных для первого знакомства.

Построение сглаживающих поверхностей, как и в рассмотренном выше случае кривых, удобно начать с описания уравнений элементарных фрагментов.

Ограничившись бикубическим случаем (именно такие сплайновые поверхности наиболее часто используются в задачах компьютерной графики), когда функциональные коэффициенты $a_i(u)$ и $b_j(v)$ представляют собой многочлены третьей степени относительно соответствующих переменных (кубические многочлены), запишем для заданного набора из 16 точек

V_{ij}, 1 = 0,1,2,3, j = 0,1,2,3,

параметрические уравнения элементарных фрагментов некоторых поверхностей, считая для простоты, что область изменения параметров u и v представляет собой единичный квадрат (рис. 21).

Начнем с элементарной бикубической поверхности Безье. Параметрические уравнения фрагмента этой поверхности имеют следующий вид:

$$r(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} C_3^i C_3^j u^i (1-u)^{3-i} v^j (1-v)^{3-j} V_{ij},$$

$0 \le u \le 1, 0 \le v \le 1.$

или, в матричной форме:

$$\begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix} = \begin{pmatrix} 1 & u & u^2 & u^3 \end{pmatrix} M^T \begin{pmatrix} V_{00} & V_{01} & V_{02} & V_{03} \\ V_{10} & V_{11} & V_{12} & V_{13} \\ V_{20} & V_{21} & V_{22} & V_{23} \\ V_{30} & V_{31} & V_{32} & V_{33} \end{pmatrix} M \begin{pmatrix} 1 \\ v \\ v^2 \\ v^3 \end{pmatrix}$$

Здесь

$$M = \begin{pmatrix} 1 & -3 & 3 & 1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

базисная матрица Безье; знаком T обозначена операция транспонирования.

Элементарная бикубическая поверхность Безье наследует многие свойства элементарной кубической кривой Безье:

- лежит в выпуклой оболочке порождающих ее точек;
- является гладкой поверхностью;
- упираясь в точки V_{00}, V_{30}, V_{30},V_{33}, касается исходящих из них отрезков контрольного графа заданного набора (рис. 23).

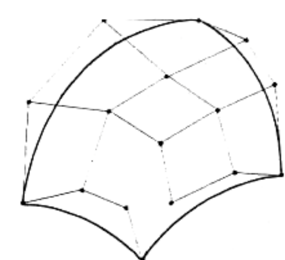


Рис. 23

Из элементарных вырезков поверхностей Безье подобно тому, как это делалось в одномерном случае, можно строить составные поверхности. Поговорим немного об условиях гладкости таких составных бикубических поверхностей Безье.

Пусть

r = $r^{(1)}$(u,v), $0 \le u \le 1$, $0 \le v \le 1$,

и

r = $r^{(2)}$(u,v), $0 \le u \le 1$, $0 \le v \le 1$, -

параметрические уравнения двух элементарных бикубических поверхностей Безье, порожденных наборами

$V_{ij}^{(1)}$, i = 0,1,2,3, j = 0,1,2,3,

и

$V_{ij}^{(2)}$, i = 0,1,2,3, j = 0,1,2,3,

соответственно и такими, что

$V_{ij}^{(1)} = V_{ij}^{(2)}$, j = 0,1, 2, 3.

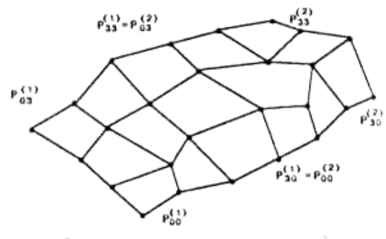


Рис. 24

Последнее означает, что эти элементарные фрагменты имеют общую граничную кривую. Поверхность, составленная из этих двух фрагментов, будет иметь непрерывную касательную плоскость, если каждая тройка точек вида

$V_{2j}^{(1)}, V_{3j}^{(1)} = V_{0j}^{(2)}, V_{ij}^{(2)}$

лежит на одной прямой и, кроме того, отношения

$$\frac{V_{2j}^{(1)}V_{3j}^{(1)}}{V_{0j}^{(2)}V_{1j}^{(2)}}$$

не зависят от номера j (рис. 24).

```
// Bezier.cpp
#include "Vector.h"
double B ( int i, double t )
{
   double s = 1.0 - t;
   switch( i ) {
   case 0:    return s * s * s;
   case 1:    return 3 * t * s * s;
   case 2:    return 3 * t * t * s;
   case 3:    return t * t * t; } }
   Vector Bezier_3_3(Vector p[], int n, double u, double v, int i, int j)
   {
      Vector t ( 0 );
      for ( int k = 0; k < 4; k++ )
      {
      Vector  s ( 0 );
      for ( int l = 0; l < 4; l++ )
         s += B ( l, v ) * p [ (i+k)*n + j+l ]; t += B ( k, u ) * s;
      }
   return t;
}
```

Векторное параметрическое уравнение элементарного фрагмента бикубической B-сплайновой поверхности, порожденной набором 16 точек

V_{ij}, i = 0,l,2, 3, j =0, 1,2,3,

имеет следующий вид:

$$r(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} n_i(u) n_j(v) V_{ij}, 0 \le u \le 1, 0 \le v \le 1$$

(функциональные коэффициенты n_0, n_1, n_2, n_3 те же, что и выше) или, в матричной форме,

$$r(u,v) = U^T M^T W M V, \ 0 \le u, v \le 1,$$

где

$$r(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}, U = \begin{pmatrix} 1 \\ u \\ u^2 \\ u^3 \end{pmatrix}, V = \begin{pmatrix} 1 \\ v \\ v^2 \\ v^3 \end{pmatrix},$$

$$W = \begin{pmatrix} V_{00} & V_{01} & V_{02} & V_{03} \\ V_{10} & V_{11} & V_{12} & V_{13} \\ V_{20} & V_{21} & V_{22} & V_{23} \\ V_{30} & V_{31} & V_{32} & V_{33} \end{pmatrix}$$

Здесь M - базисная матрица кубического B-сплайна.

Как и бикубическая поверхность Безье, элементарная бикубическая B-сплайновая поверхность наследует многие свойства элементарной кубической B-сплайновой кривой:

- является гладкой;
- лежит в выпуклой оболочке порождающих ее 16 вершин;
- повторяет" контрольную многогранную поверхность.

Построение составной бикубической B-сплайновой поверхности (обладающей весьма привлекательными геометрическими свойствами) на прямоугольнике

$[0,m]\times[0,n]$

с равномерными узлами (i, j), i = 0,1,..., m -1, m, j = 0, 1,..., n -1, n, проводится во многом подобно тому, как это делается в одномерно случае.

Разумеется, существуют и весьма эффективно используются двумерные аналоги и рациональных B-сплайновых кривых (как на равномерной сетке, так и на неравномерной (NURBS), и бета-сплайновых кривых.

Выпишем, например, векторное уравнение элементарной бета-сплайновой поверхности - (к, 1)-вырезка для заданного набора$(m+1)(n+1)$вершин. Имеем:

$$r_{k1} = r_{k1}(u,v) = \sum_{i=-2}^{1}\sum_{j=-2}^{1} b_i(u) b_j(v) V_{i+k,\,j+1}, 0 \le u, v \le 1.$$

```
// Beta.cpp
#include <math.h>
#include "Vector.h"
static double beta1, beta2;
static double b12, b13, b22, b23;
static double delta, d;
double b ( int i, double t )
{
   double s = 1.0 - t;
   double t2 = t * t;
   double t3 = t2 * t;
   switch ( i )
   {
      case 0:   return 2 * b13 * d * s * s * s;
      case 1:   return d * (2*b13*t4t2-3*t+3)+2*b12*(t3-3*t2+2)+
             2 * beta1 * (t3-3*t+2)+beta2*(2*t3-3*t2+1));
      case 2:   return d*(2*b12*t2*(-t+3)+2*beta1*t*(-t2+3)+
      beta2*t2-(-2*t+3)+2*(-t3+1));
      case 3:   return 2 * t3 * d;
   }
}

Vector Beta_3_3(double b1, double b2, Vector p[], int n,
double u, double v, int i, int j )
{
   Vector t ( 0 );
   beta1 = b1;
   beta2 = b2;
   b12   = beta1 * beta1;
   b13   = b12 * beta1;
   b22   = beta2 * beta2;
   b23   = b22 * beta2:
   delta = 2 * b13 + 4 * b12 + 4 * beta1 + beta2 + 2;
   0     = 1.0 / delta;
   for(int k = 0; k < 4; k++ ) {
     Vector s ( 0 );
     for (int l = 0; 1<4; l++)
         s += p [(i+k)*n+j+l] * b (l, v);
     t += s * b ( k, u );
   }
   return t;
}
```

Мы остановились в этой главе лишь на некоторых простых способах построения плавно изменяющихся кривых и поверхностей. Вводный характер книги и жесткие ограничения на ее объем не позволяют говорить об этом более подробно. К сказанному следует также добавить, что построение искривленных пространственных объектов является действительно непростой задачей. Она требует достаточно развитого пространственного воображения и почти постоянной готовности к встрече с вещами неожиданными. Хотя и объяснимыми, но не сразу, а после заметных усилий. Тем не менее мы стремились к тому,

чтобы по отобранному материалу и программным реализациям представленных алгоритмов у читателя сложилось в целом правильное начальное представление о геометрических сплайнах и том месте, которое они занимают в компьютерной графике. По нашему мнению, даже небольшая самостоятельная попытка компьютерной реализации высказанных здесь сравнительно несложных геометрических соображений будет, несомненно, полезна в освоении практически неисчерпаемых возможностей компьютерной графики.

ОСНОВЫ МЕТОДА ТРАССИРОВКИ ЛУЧЕЙ

Поистине он (Леонардо) захотел от живописи удивительной, вещи и, думаю, достиг желаемого: он захотел сделать живопись трехмерной, и третьим измерением является здесь время. Плоскость картины протяженна не только иллюзорно-пространственно, но и действительно протяженна во времени, но мы видим и воспринимаем время не как обычно, т есть в виде последовательных движений и изменений, а так словно прошлое и будущее зазвучали в некотором пространственном настоящем и вместе с ним.

А. Ф. Лосев

Одним из наиболее распространенных и наглядных методов построения реалистических изображений является метод трассировки лучей, позволяющий строить фотореалистические изображения сложных сцен с учетом таких эффектов, как отражение и преломление. Отличительной чертой метода является его крайняя простота и наглядность.

Для того чтобы понять, каким образом можно построить искусственное изображение сцены, рассмотрим, каким путем возникает изображение реальной сцены в глазе наблюдателя.

Пусть задана реальная сцена (рис. 1), состоящая из источника света и ряда объектов.

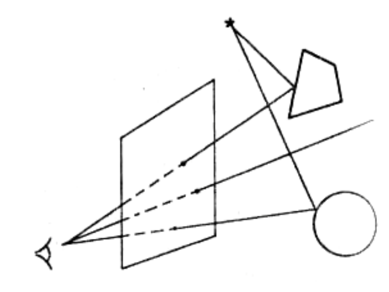


Рис 1

Весь свет начинает свой путь из источника и распространяется от него по прямолинейным траекториям до попадания на объекты сцены. Попав на какой-либо объект сцены, луч света может преломиться и уйти внутрь объекта или отразиться (рассеяться). Отразившись от объекта, луч света опять распространяется прямолинейно до попадания на следующий объект, и так далее. Часть лучей в конце концов попадает в глаз наблюдателя, формируя изображение сцены на сетчатке его. Поместим перед глазом воображаемую картинную плоскость (экран) и будем считать, что изображение формируется на этой плоскости. Каждый луч, попадающий в глаз, проходит через некоторую точку экрана, формируя там изображение. Тем самым для построения изображения достаточно проследить весь путь распространения света, начиная от его источника.

Выпустим из каждого источника света пучок лучей во все стороны и мысленно проследим (оттрассируем) дальнейшее распространение каждого из них до тех пор, пока либо он не попадет в глаз наблюдателя, либо не покинет сцену. При попадании луча на границу объекта выпускаем из точки попадания отраженный и преломленный лучи и отслеживаем их и все порожденные ими лучи.

Описанный процесс называется прямой трассировкой лучей. В результате его выполнения можно получить изображение сцены, однако он требует огромных вычислительных затрат.

Основным недостатком прямой трассировки лучей является то обстоятельство, что в получаемое изображение сколько-нибудь существенный вклад вносит лишь очень небольшая часть трассируемых лучей. Тем самым при реализации этого метода основная часть работы оказывается проделанной впустую.

Чтобы избежать этого, попытаемся вместо трассирования всех лучей отслеживать лишь те лучи, которые вносят заметный вклад в строящееся изображение. Ясно, что это те лучи, которые попадают в глаз наблюдателя.

Для определения освещенности (цвета) точки экрана можно проследить путь, по которому мог пройти луч света, попавший в эту точку и сформировавший там изображение. Очевидно, что таким путем является путь луча, выходящего из глаза наблюдателя и проходящего через соответствующую точку экрана. Будем идти вдоль этого луча от глаза до точки ближайшего

пересечения с каким-либо объектом сцены (при этом мы будем перемешаться в направлении, обратном направлению распространения света). Цвет соответствующей точки экрана будет определяться долей световой энергии, попадающей в эту точку и покидающей ее в направлении глаза. Для определения этой энергии необходимо найти освещенность точки объекта, для чего из нее выпускаются лучи в тех направлениях, из которых может прийти энергия. Это, в свою очередь, может привести к определению точек пересечения соответствующих лучей с объектами сцены, выпускания новых лучей и так далее.

Описанный процесс называется обратной трассировкой лучей или просто трассировкой лучей. Именно этот метод и будет рассматриваться далее.

Ключевая задача метода трассировки лучей - определение освещенности произвольной точки объекта и той части световой энергии, которая уходит в заданном направлении. Эта энергия складывается из двух частей - непосредственной (первичной) освещенности, то есть энергии, непосредственно получаемой от источников света, и вторичной освещенности, то есть энергии, идущей от других объектов.

Конечно, такое деление носит условный характер.

Ясно, что непосредственная освещенность вносит существенно больший вклад в изображение. Поэтому обычно первичная и вторичная освещенность рассматриваются по-разному.

Отбрасываемая энергия состоит из той энергии, которая отражается и преломляется в заданном направлении.

Для эффективного пользования методом трассировки лучей необходимо понимание физики процессов отражения и преломления.

Немного физики

Рассмотрим процесс распространения света.

Известно, что свет можно рассматривать и как поток частиц, распространяющихся по прямолинейным траекториям, и как электромагнитную волну, распространяющуюся в пространстве. При этом интенсивность света определяется амплитудой волны, а его цвет - частотой или длиной волны λ. Сам процесс распространения света описываете уравнениями Максвелла.

Произвольный луч света можно рассматривать как сумму волн различными длинами, распространяющихся в одном направлении. Вклад волны с длиной λ определяется функцией $I(\lambda)$, называемой спектральной кривой (характеристикой) данного луча света.

В действительности один и тот же воспринимаемый глазом цвет может вызываться бесконечным количеством различных источников света с различными спектральными кривыми $I(\lambda)$. Поэтому при исследовании обычно ограничиваются конечным набором значений λ, например для чистых красного, зеленого и синего цветов, и представляют все цвета в виде линейной комбинации этих базовых цветов.

Процесс распространения света распадается на две части - распространение света в однородной среде и взаимодействие света с границей раздела двух сред.

Распространение света в однородной среде происходит вдоль прямолинейной траектории с постоянной скоростью. Отношение скорости распространения света в вакууме к этой скорости называется коэффициентом преломления (индексом рефракции) среды η. Обычно этот коэффициент зависит от длины волны λ.

При распространении света в среде может иметь место экспоненциальное затухание с коэффициентом $e^{-\beta l}$, где l - расстояние, пройденное лучом в среде, а β - коэффициент затухания.

При взаимодействии с границей двух сред происходит отражение и преломление света. Рассмотрим несколько идеальных моделей. в каждой из которых границей раздела сред является плоскость.

1. Зеркальное отражение

Отраженный луч падает в точку P в направлении i и отражается в направлении, задаваемом вектором r, определяемым следующим законом: вектор r лежит в той же плоскости, что и вектор i и единичный вектор внешней нормали к поверхности n, а угол падения θ_i равен углу отражения θ_r (рис. 2).

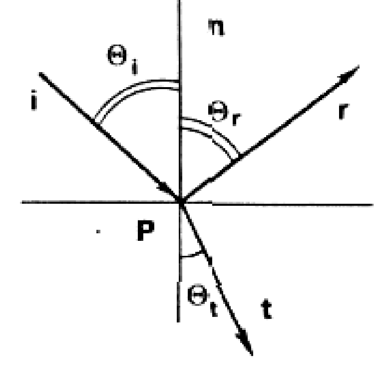


Рис. 2

Будем считать все векторы единичными. Тогда из первого условия следует, что вектор i равен линейной комбинации векторов i и n, то есть

$r = \alpha i + \beta n.$ (1)

Так как $\theta_i = \theta_r$,

то $(-i,n) = \cos\theta_i = \cos\theta_r = (r, n)$.

Отсюда легко получается

$r = i - 2(i,n)n.$ (2)

Несложно убедиться, что вектор, задаваемый соотношением (2), является единичным.

2. Диффузное отражение

Идеальное диффузное отражение описывается законом Ламберта, согласно которому падающий свет рассеивается во все стороны с одинаковой интенсивностью. Таким образом не существует однозначно определенного направления, в которым бы отражается падающий луч, все направления равноправны и освещенность точки пропорциональна только доле площади, видимой от источника, то есть (i, n).

3. Идеальное преломление

Луч, падающий в точку P в направлении вектора i, преломляется внутрь второй среды в направлении вектора t (рис. 2). Преломление подчиняется закону Снеллиуса, согласно которому векторы i, n и t лежат в одной плоскости и для углов справедливо соотношение

$\eta_i \sin\theta_i = \eta_t \sin\theta_t.$ (3)

Найдем для вектора t явное выражение. Этот вектор можно представить в следующем виде:

$t = \alpha i + \beta n.$

Соотношение (3) можно переписать так:

$\sin\theta_t = \eta \sin\theta_i,$ (4)

где $\eta = \dfrac{\eta_i}{\eta_t}$ (5)

Тогда

$\eta^2 \sin^2\theta_i = \sin^2\theta_t$

или

$\eta^2(1-\cos^2\theta_i) = 1-\cos^2\theta_t.$ (6)

Так как

$\cos\theta_i = (-i,n), \cos\theta_t = (-t,n),$

то

$$\alpha^2(i,n)^2 + 2\alpha\beta(i,n) + \beta^2 = 1 + \eta^2((i,n)^2 - 1. \quad (7)$$

Из условия нормировки вектора t имеем

$$\|t\|^2 = (t,t) = \alpha^2 + 2\alpha\beta(i,n) + \beta^2 = 1. \quad (8)$$

Вычитая это соотношение из равенства (7), имеем:

$$\alpha^2((i,n)^2 - 1) = \eta^2((i,n)^2 - 1), \quad (9)$$

откуда a = ±μ.

Из физических соображений следует, что a = η.

Второй параметр определяется из уравнения

$$\beta^2 + 2\beta\eta(i,n) + \eta^2 - 1 = 0, \quad (10)$$

дискриминант которого равен

$$D = 4\{1 + \eta^2((i,n)^2 - 1\}. \quad (11)$$

Решение этого уравнения задается формулой

$$\beta = \frac{-2\eta \pm 2\sqrt{1+\eta^2((i,n)^2 - 1)}}{2}, \quad (12)$$

и, значит, вектор

$$t = \eta i + \left\{\eta C_i - \sqrt{1+\eta^2(C_i^2 - 1)}\right\} n, \quad (13)$$

где $C_i = \cos\theta_i = -(i,n)$. (14)

При этом случай, когда выражение над корнем отрицательно $(1+\eta^2(C_i^2 - 1) < 0$ соответствует так называемому полному внутреннему отражению, когда вся световая энергия отражается от границы раздела сред и преломления фактически не происходит.

4. Диффузное преломление

Диффузное преломление полностью аналогично диффузному отражению, при этом преломленный луч идет по всем направлениям t : (t, n) < 0 с одинаковой интенсивностью.

Рассмотрим теперь распределение энергии при отражении и преломлении. Из курса физики известно, что доля отраженной энергии задается коэффициентами Френеля

$$F_r(\lambda,\theta) = \frac{1}{2}\left\{\left(\frac{\cos\theta_i - \eta\cos\theta_t}{\cos\theta_i + \eta\cos\theta_t}\right)^2 + \left(\frac{\eta\cos\theta_i - \cos\theta_t}{\eta\cos\theta_i + \cos\theta_t}\right)^2\right\}. \quad (15)$$

Существует другая форма записи этих соотношений:

$$F_r(\lambda,\theta) = \frac{1}{2}\left(\frac{c-g}{c+g}\right)\left\{1 + \left(\frac{c(c+g)-1}{c(c-g)-1}\right)^2\right\}, \quad (16)$$

где c = $\cos\theta_i$; $g = \sqrt{\eta^2 + c^2 - 1} = \eta\cos\theta_t$ (17)

Формула (15) верна для диэлектрических материалов.

Для проводников обычно используется следующая формула

$$F_r = \frac{1}{2}\left\{\left(\frac{(\eta_t^2 + k_t^2)\cos^2\theta_i - 2\eta_t\cos\theta_i + 1}{(\eta_t^2 + k_t^2)\cos^2\theta_i + 2\eta_t\cos\theta_i + 1}\right)^2 + \left(\frac{(\eta_t^2 + k_t^2) - 2\eta_t\cos\theta_i + \cos^2\theta_i}{(\eta_t^2 + k_t^2) + 2\eta_t\cos\theta_i + \cos^2\theta_i}\right)^2\right\}, \quad (18)$$

где k_t - индекс поглощения.

Ясно, что все рассмотренные примеры являются идеализациями. На самом деле нет ни идеальных зеркал, ни идеально гладких поверхностей.

На практике обычно считают, что поверхность состоит из множества случайно ориентированных плоских идеальных микрозеркал (микрограней), с заданным законом распределения (рис. 3).

Пусть n - нормаль к поверхности (ее средней линии), h - вектор нормали к микрограни и α - угол между ними,

α = arccos(n,h).

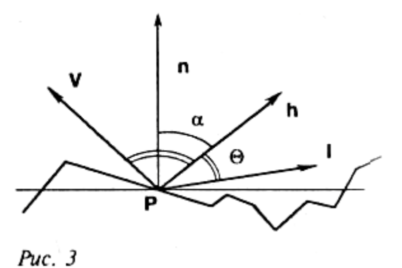


Рис. 3

Поверхность будем описывать с помощью функции D(α), задающей плотность распределения случайной величины α (для идеально гладкой поверхности функция D(α) совпадает с δ-функцией Дирака).

Существует несколько распространенных моделей для функции D(α):

Гауссовское распределение

$$D(\alpha) = Ce^{-\left(\frac{\alpha}{m}\right)^2}, \quad (19)$$

распределение Бекмена

$$D(\alpha) = \frac{l}{4\pi m^2 \cos^4 \alpha} e^{-\left(\frac{tg\alpha}{m}\right)^2}. \quad (20)$$

В этих моделях m характеризует степень неровности поверхности -чем меньше т, тем более гладкой является поверхность.

Рассмотрим отражение луча света, падающего в точку P вдоль направления, задаваемого вектором 1. Пусть n - вектор внешней нормали к поверхности. Поскольку микрограни распределены случайным образом, то отраженный луч может уйти практически в любую сторону. Определим долю энергии, уходящей в заданном направлении v. Для того, чтобы луч отразился в этом направлении, необходимо, чтобы он попал на микрогрань, нормаль h к которой удовлетворяет соотношению

$$h = \frac{l+v}{\|l+v\|}. \quad (21)$$

Доля энергии, которая отразится от микрограни, определяется коэффициентом Френеля $F_r(\lambda,\theta)$, где

θ = arccos(h,v) = arccos(h, l).

Если поверхность состоит из множества микрограней, начинает сказываться затеняющее влияние соседних граней, которое обычно описывается с помощью следующей функции:

$$G = \min\left(l, \frac{2(n,h)(n,v)}{(v,h)}, \frac{2(n,h)(n,l)}{(v,h)}\right). \quad (22)$$

В этом случае интересующая нас доля энергии задается формулой

$$\frac{F_r(\lambda,\theta)D(\alpha)G(n,v,l)}{(n,l)(n,v)} \quad (23)$$

Совершенно аналогично рассматривается преломление света поверхностью, состоящей из микрозеркал.

С использованием соотношения (23) можно построить формулу, полностью описывающую энергию (и отраженную, и преломленную) в заданном направлении. Для этого необходимо выпустить лучи во все возможные стороны и вычислить приходящую оттуда энергию, то есть в качестве вектора l можно рассматривать любой единичный вектор. Ясно, что на практике это невозможно.

Поэтому желательно взять алгоритм, отслеживающий лишь конечное число направлений, вносящих в искомую величину наибольший вклад.

Основная модель трассировки лучей

Введем некоторые ограничения на рассматриваемую сцену:

- будем рассматривать только точечные источники света;
- при трассировании преломленного луча будем игнорировать зависимость его направления от длины волны;
- будем считать освещенность объекта состоящей из диффузной и зеркальной частей (с заданными весами).

Для определения освещенности точки P определим сначала непосредственную освещенность этой точки от источников света (выпустив из нее лучи ко всем источникам).

Для определения вторичной освещенности выпустим из точки P один луч для отраженного направления и один луч для преломленного. Тем самым для определения освещенности точки необходимо будет отслеживать лишь небольшое количество лучей.

При этом неидеально зеркальное отражение лучей, идущих от других объектов, игнорируется.

Обычно для компенсации всех таких неучитываемых величин вводится так называемое фоновое освещение - равномерное освещение со всех сторон, которое ни от чего не зависит и не затеняется.

Тогда энергия, покидающая точку P в заданном направлении, задается следующей формулой:

$$I(\lambda) = K_a I_a(\lambda) C(\lambda) + K_d C(\lambda) \sum_i I_{l_i}(\lambda)(n, l_i) + K_s \sum_i I_{l_i}(\lambda) \frac{F_r(\lambda, \theta) D(\alpha) G}{(n, l_i)(n, v)} + \qquad (24)$$

$$+ K_s I_r(\lambda) F_r(\lambda, \theta_r) e^{-\beta_r d_r} + K_t I_t(\lambda)(1 - F_r(\lambda, \theta_t)) e^{-\beta_t d_r},$$

где

$I_a(\lambda)$ - интенсивность фонового освещения;

$I_{li}(\lambda)$ - интенсивность i-го источника света;

$I_r(\lambda)$ - интенсивность, приходящая по отраженному лучу;

$I_t(\lambda)$ - освещенность, приносимая преломленным лучом;

$C(\lambda)$ - цвет в точке P;

K_a - коэффициент фонового освещения;

K_d - коэффициент диффузного освещение;

K_s - коэффициент зеркального освещения;

K_t - вклад преломленного луча;

n - вектор внешней нормали в точке P,

l_i - единичный вектор направления из точки P на i-й источник света;

θ_r - угол отражения (для отраженного луча);

θ_t - угол преломления;

d_r - расстояние, пройденное отраженным лучом;

d_t - расстояние, пройденное преломленным лучом;

β_r - коэффициент ослабления для отраженного луча;

β_t - коэффициент ослабления для преломленного луча.

К сожалению, эта модель, хотя и является достаточно физически корректной, слишком сложна для практического воплощения. Поэтому часто используются более простые модели, например модель Холла:

$$I(\lambda) = K_a I_a(\lambda) C(\lambda) + K_d C(\lambda) \sum_i I_{l_i}(\lambda)(n, l_i) + \\ + K_s \sum_i I_{l_i}(\lambda) F_r(\lambda, \theta_i)(n, h_i)^p + K_s I_r(\lambda) F_r(\lambda, \theta_r) e^{-\beta_r d_r} + \\ + K_t I_t(\lambda)(1 - F_r(\lambda, \theta_t)) e^{-\beta_t d_r}. \quad (25)$$

Несмотря на то, что коэффициенты Френеля заметно влияют на степень реалистичности изображения, на практике их применяют очень редко. Дело в том, что их использование наталкивается на ряд серьезных препятствий, одним из которых является сложность вычисления, а другим - отсутствие точной информации о зависимости величин, входящих в состав формулы, от длины волны λ.Существуют определенные методы интерполяции коэффициентов Френеля (например, линейная), которые в сочетании с табличным способом задания способны заметно ускорить процесс их вычисления, однако их рассмотрение выходит за рамки данной книги. Далее будет рассматриваться модель Уиттеда:

$$I(\lambda) = K_a I_a(\lambda) C(\lambda) + K_d C(\lambda) \sum_i I_{l_i}(\lambda)(n, l_i) + \\ + K_s \sum_i I_{l_i}(\lambda)(n, h_i)^k + K_s I_r(\lambda) e^{-\beta_r d_r} + K_t I_t(\lambda) e^{-\beta_t d_r}. \quad (26)$$

Замечание

Часто вместо члена $(n, h)^p$ используется $(r, l)^p$.

Тем самым мы приходим к следующей схеме трассировки лучей: через каждый пиксел экрана луч трассируется до ближайшего пересечения с объектами сцены. Из точки пересечения выпускаются лучи ко всем источникам света для проверки их видимости и определения непосредственной освещенности точки пересечения. Выпускаются также отраженный и преломленный лучи, которые, трассируются, в свою очередь, до ближайшего пересечения с объектами сцены, и так далее. Получается рекурсивный алгоритм трассировки.

В качестве критерия остановки обычно используется отсечение по глубине (не более заданного количества уровней рекурсии) и по весу (чем дальше, тем меньше вклад каждого луча в итоговой цвет пиксела, и, как только этот вклад опускается ниже некоторого порогового значения, дальнейшая трассировка этого луча прекращается).

Рассмотрим простейшую реализацию этой модели.

Материал, в котором распространяется луч, будем описывать структурой Medium, состоящей из двух величин - коэффициента преломления nRefr и коэффициента поглощения Betta.

Свойства поверхности зададим следующими величинами (структура SurfaceData): Ka, Kd, Ks, Kr и Kt - веса фоновой, диффузной, зеркальной, отраженной и преломленной освещенности, Color - цвет, Med - материал, из которого состоит объект и степень p. Введем сюда также вектор нормали n.

Абстрактный источник света LightSource содержит свой цвет и виртуальный метод Shadow, определяющий для произвольно заданной точки направление на источник и долю энергии, доходящей до заданной точки (с учетом затенения другими объектами и зависимости от расстояния).

Модель абстрактного объекта (класс GObject) содержит стандартный используемый материал DefMaterial, метод FindTexture, служащий для определения свойств поверхности объекта в заданной точке Также объект содержит виртуальные методы Intersect для определения ближайшей точки пересечения луча с объектом и для вычисления расстояния до точки пересечения и метод FindNormal для определения нормали в произвольной заданной точке границы объекта.

Для представления всей сцены используется класс Environment, содержащий массивы ссылок на все используемые в сцене источники света и объекты. Этот класс содержит также метод Intersect, служащий для определения ближайшей точки пересечения луча с объектами сцены, и метод Shade Background, служащий для определения энергии, приносимой лучом, не попавшим ни в один объект сцены.

Для задания положения наблюдателя (камеры) служит функция SetCamera, задающая положение наблюдателя, направление обзора и направление верха.

Упомянутые объекты и процедуры содержатся в файлах Tracer.h и Tracer.cpp, приведенных ниже.

```
// File Tracer.h
#ifndef  __TRACER__
#define  __TRACER__

#include <math.h>
#include <stdlib.h>
#include "Vector.h"

#define  MAX_LIGHTS  10
#define  MAX_SOLIDS  100
#define  INFINITY    30000

struct   Medium                  // main properties of the medium
{
   doublenRefr;            // refraction coefficient
   doubleBetta;            // attenuation coefficient
};

struct   SurfaceData             // surface charactericstics at a given point
{
   doubleKa;               // ambient light coefficient
   doubleKd;               // diffuse light coefficient
   doubleKs;               // specular light coefficient
   doubleKr;               // reflected ray coefficient
   doubleKt;               // transparent light coefficient
   VectorColor;            // object's color
   MediumMed;              // medium of the object
   int   p;                // Phong's coeff.
   Vectorn;                // normal at a given point
};

class    LightSource             // model of an abstract light source
{
public:
   VectorColor;

   LightSource () { Color = 1; };
   virtual     ~LightSource () {};     // force virtual destructor in all descendants

   virtual     doubleShadow ( Vector&, Vector& ) = 0;
};

class    GObject                       // model of an abstract geometric object
{
public:
   SurfaceData DefMaterial;// default material ( if no texture specified )

   GObject () {};
   virtual     ~GObject () {};         // force virtual destructor in all descendants

   void        FindTexture ( Vector& p, SurfaceData& t ) { t = DefMaterial; t.n =
FindNormal ( p ); };
   virtual     int   Intersect ( Ray&, double& ) = 0;
```

```
      virtual     VectorFindNormal ( Vector& ) = 0;
   };

   class    Environment             // simplest model of environment
   {
   public:
      LightSource * Light [MAX_LIGHTS];
      int         LightsCount;
      GObject     * Solid [MAX_SOLIDS];
      int         SolidsCount;

      Environment () { LightsCount = SolidsCount = 0; };
      ~Environment ();

      void          Add ( LightSource * );
      void          Add ( GObject * );
      virtual     GObject * Intersect ( Ray&, double& );
      virtual     Vector  ShadeBackground ( Ray& );
   };

   /////////////////////////////  Globals  ////////////////////////////////////

   extern   Vector      Eye;                    // camera position
   extern   Vector      EyeDir;                       // camera viewing direction
   extern   Vector      Vx, Vy;                       // image plane basis ( Vx - hor, Vy
- vert )
   extern   Medium      Air;                    // basic materials
   extern   Medium      Glass;
   extern   int         Level;                  // current recusion level
   extern   double      Threshold;
   extern   int         MaxLevel;               // max. levels of recursion
   extern   Vector      Ambient;                // ambient light intensity
   extern   Vector      Background;
   extern   Environment   * Scene;
   extern   long        TotalRays;

   ////////////////////////// Function definitions ////////////////////////

   void     Camera ( double, double, Ray& );                      // get ray for given
sample point
   void     SetCamera ( Vector&, Vector&, Vector& );         // set new camera parameters
   Vector   Trace  ( Medium&, double, Ray& );                // trace a ray in given
medium, ret. ray length
   Vector   Shade  ( Medium&, double, Vector&, Vector&, GObject * ); // compute light at
given point
   double   SawWave ( double );

   inline   doubleSineWave ( double x )
   {
      return 0.5 * ( 1.0 + sin ( x ) );
   }

   inline   doubleMod ( double x, double y )
   {
      if ( ( x = fmod ( x, y ) ) < 0 )
            return x + y;
      else
            return x;
   }

   inline   doubleRnd ()
   {
      return ( (double) rand () ) / (double) RAND_MAX;
   }

   #endif
```

```
//File Tracer.cpp
#include <alloc.h>
#include <stdlib.h>

#include "Tracer.h"

/////////////////////////////  Globals  ///////////////////////////////////

Vector        Eye ( 0, 0, 0 );         // camera position
Vector        EyeDir ( 0, 0, 1 );            // viewing directing ( normal to image
plane )
Vector        Vx ( 1, 0, 0 );                // image plane basis - horizontal vector
Vector        Vy ( 0, 1, 0 );                // vertical vector
Vector        Ambient ( 1.0 );         // ambient light intensity
Vector        Background ( 0.0, 0.05, 0.05 );      // background color
Medium        Air           = { 1, 0 };      // basic mediums : Air
Medium        Glass         = { 1.5, 0 };    // glass
int           Level         = 0;       // current recusion level
double        Threshold     = 0.01;          // accuracy of computations
int           MaxLevel      = 10;            // mx. levels of recursion
Environment  * Scene;
long          TotalRays     = 0l;

////////////////////////// Environment methods ////////////////////////////

Environment :: ~Environment ()
{
  int   i;
                                 //delete all contained objects
  for ( i = 0; i < LightsCount; i++ )
        delete Light [i];

  for ( i = 0; i < SolidsCount; i++ )
        delete Solid [i];
}

void     Environment :: Add ( LightSource * l )
{
  if ( LightsCount < MAX_LIGHTS - 1 )
        Light [LightsCount++] = l;
}

void     Environment :: Add ( GObject * o )
{
  if ( SolidsCount < MAX_SOLIDS - 1 )
        Solid [SolidsCount++] = o;
}

//////////////////////////////////////////////////
// find closest intersection with scene objects //
//////////////////////////////////////////////////

GObject * Environment :: Intersect ( Ray& ray, double& t )
{
  GObject * ClosestObj  = NULL;
  double  ClosestDist = INFINITY;

  for ( int i = 0; i < SolidsCount; i++ )                // check every object
        if ( Solid [i] -> Intersect ( ray, t ) )   // if intersect then check distance
              if ( t < ClosestDist )
              {
                    ClosestDist = t;
                    ClosestObj  = Solid [i];
              }
```

```
      t = ClosestDist;

      return ClosestObj;
   }
   #pragma  argsused                 // turn off parameter no used warning
   Vector   Environment :: ShadeBackground ( Ray& ray )
   {
      return Background;
   }

   ////////////////////////// Functions ///////////////////

   void     SetCamera ( Vector& Org, Vector& Dir, Vector& UpDir )
   {
      Eye    = Org;                        // eye point
      EyeDir = Dir;                        // viewing direction
      Vx     = Normalize ( UpDir ^ Dir );  // build orthogonal basis of image plane
      Vy     = Normalize ( Dir ^ Vx );     // EyeDir orthgonal to this basic ( image plane
)
   }

   ///////////////////////////////////////////////////////
   // get a pixel ray for a given screen point ( x, y ) //
   ///////////////////////////////////////////////////////

   void     Camera ( double x, double y, Ray& ray )
   {
      ray.Org = Eye;
      ray.Dir = Normalize ( EyeDir + Vx * x + Vy * y );
   }

   /////////////////////////////////////////
   // Trace a given ray through the scene //
   /////////////////////////////////////////

   Vector   Trace ( Medium& CurMed, double Weight, Ray& ray )
   {
      GObject * Obj;
      double  t = INFINITY;
      Vector  Color;

      Level++;
      TotalRays ++;

      if ( ( Obj = Scene -> Intersect ( ray, t ) ) != NULL ) // ray hit some object
      {
            Color = Shade ( CurMed, Weight, ray.Point ( t ), ray.Dir, Obj );

            if ( CurMed.Betta > Threshold )
                  Color *= exp ( -t * CurMed.Betta );
      }
      else
            Color = Scene -> ShadeBackground ( ray );

      Level--;

      return Color;
   }

   ////////////////////////////////////////////////////////////
   // compute light coming from point p in the direction View //
   //       using Whitted's illumination model                //
   ////////////////////////////////////////////////////////////
```

```
   Vector   Shade ( Medium& CurMed, double Weight, Vector& p, Vector& View, GObject * Obj
)
   {
      SurfaceData txt;
      Ray         ray;
      Vector      Color;
      Vector      l;                            // light vector
      double      Sh;                           // light shadow coeff.
      Vector      h;                            // vector between -View and light vector
      double      ln, vn;
      int         Entering = 1;                       // flag whether we're entering or
leaving object

      Obj -> FindTexture ( p, txt );

      if ( ( vn = View & txt.n ) > 0 )              // force ( -View, n ) > 0
      {
            txt.n    = -txt.n;
            vn       = -vn;
            Entering = 0;
      }

      ray.Org = p;                              // since all rays will be cast from here
      Color   = Ambient * txt.Color * txt.Ka;               // get ambient light

      for ( int i = 0; i < Scene -> LightsCount; i++ )
            if ( ( Sh = Scene -> Light [i] -> Shadow ( p, l ) ) > Threshold )
                  if ( ( ln = l & txt.n ) > Threshold )      // if light is visible
                  {
                        if ( txt.Kd > Threshold )     // compute direct diffuse light
                              Color += Scene -> Light [i] -> Color * txt.Color * ( txt.Kd
* Sh * ln );

                        if ( txt.Ks > Threshold )     // compute direct specular light,
via Phong shading
                        {                       // compute half-vector between -View and l
                              h      = Normalize ( l - View );
                              Color += Scene -> Light [i] -> Color * ( txt.Ks *
                                     Sh * pow ( txt.n & h, txt.p));
                        }
                  }

      doublerWeight = Weight * txt.Kr;                // weight of reflected ray
      doubletWeight = Weight * txt.Kt;                // weight of transmitted ray
                                                // check for reflected ray
      if ( rWeight > Threshold && Level < MaxLevel )
      {
            ray.Dir = View - txt.n * ( 2 * vn );      // get reflected ray direction
            Color  += txt.Kr * Trace ( CurMed, rWeight, ray );
      }
                                                // check for transmitted ray
      if ( tWeight > Threshold && Level < MaxLevel )
      {                                         // relative index of refraction
            doubleEta  = CurMed.nRefr / ( Entering ? txt.Med.nRefr : Air.nRefr );
            doubleci   = - vn;                  // cosine of incident angle
            doublectSq = 1 + Eta*Eta*( ci*ci - 1 );   // squared cosine of transm. angle

            if ( ctSq > Threshold )                   // not a Total Internal Reflection
            {
                  ray.Dir = View * Eta + txt.n * ( Eta*ci - sqrt ( ctSq ) );

                  if ( Entering )                     // ray enters object ( txt.Med )
                        Color += txt.Kr * Trace ( txt.Med, tWeight, ray );
                  else                          // ray leaves object ( Air )
                        Color += txt.Kr * Trace ( Air, tWeight, ray );
            }
```

```
      }

            return Color;
   }
```

Файлы Render.cpp и Render.h содержат модуль, обеспечивающий трассировку сцены и запись построенного изображения в файл формата TGA (функция RenderScene).

```
    //File Render.h
   #ifndef  __RENDER__
   #define  __RENDER__

   #include <stdlib.h>

   void     RenderScene ( double, double, int, int, char * );
   void     DistributedRenderScene ( double, double, int, int, int, int, char * );
   void     AdaptiveDistributedRenderScene ( double, double, int, int, int, int, double,
char * );

   #endif

   //File Render.cpp
   #include <alloc.h>
   #include <conio.h>
   #include <dos.h>
   #include <fcntl.h>
   #include <io.h>
   #include <mem.h>
   #include <stdio.h>
   #include <stdlib.h>
   #include <sys\stat.h>

   #include "Tracer.h"
   #include "Draw.h"
   #include "Render.h"

   #include "Targa.h"

   long     far   * TicksPtr = ( long far * ) 0x46CL;

   void     RenderScene ( double HalfWidth, double HalfHeight, int nx, int ny, char *
PicFileName )
   {
      double    x, y;                     // sample point
      double    hx = 2.0 * HalfWidth  / nx;           // pixel width
      double    hy = 2.0 * HalfHeight / ny;           // pixel height
      Ray       ray;                      // pixel ray
      Vector    Color;
      int       i, j;
      long      Ticks = * TicksPtr;
      TargaFile * tga = new TargaFile ( PicFileName, nx, ny );
      RGB       c;

      SetMode ( 0x13 );
      SetPreviewPalette ();

      for ( i = 0, y = HalfHeight; i < ny; i++, y -= hy )
      {
            for ( j = 0, x = - HalfWidth; j < nx; j++, x += hx )
            {
                  Camera ( x, y, ray );
                  Color = Trace  ( Air, 1.0, ray );
                  Clip ( Color );
                  c.Red   = Color.x * 255;
                  c.Green = Color.y * 255;
```

```
                  c.Blue  = Color.z * 255;
                  tga -> PutPixel ( c );
                  DrawPixel ( j, i, Color );
            }
      }

      Ticks -= * TicksPtr;
      if ( Ticks < 0l )
            Ticks = -Ticks;

      delete tga;

      getch ();
      SetMode ( 0x03 );
      printf ( "\nEnd tracing." );
      DrawTargaFile ( PicFileName );
      printf ( "\nElapsed time : %d sec. ", (int)(Ticks/18) );
   }

   void     DistributedRenderScene ( double HalfWidth, double HalfHeight, int nx, int ny,
int nxSub, int nySub, char * PicFileName )
   {
      double    x, y;                     // sample point
      Ray       ray;                      // pixel ray
      double    hx = 2.0 * HalfWidth  / nx;            // pixel width
      double    hy = 2.0 * HalfHeight / ny;            // pixel height
      double    hxSub = hx / nxSub;
      double    hySub = hy / nySub;
      int       i, j;
      int       PrimarySamples = nxSub * nySub;  // # of samples taken for each pixel
      Vector    Color;
      Vector    LineBuffer [800];
      long      Ticks = * TicksPtr;
      TargaFile * tga = new TargaFile ( PicFileName, nx, ny );
      RGB       c;

      SetMode ( 0x13 );
      SetPreviewPalette ();

      for ( i = 0, y = HalfHeight; i < ny; i++, y -= hy )
      {
            for ( j = 0, x = - HalfWidth; j < nx; j++, x += hx )
            {
                  doublex1 = x - 0.5 * hx;
                  doubley1 = y - 0.5 * hy;

                  Color = 0;

                  for ( int iSub = 0; iSub < nxSub; iSub++ )
                        for ( int jSub = 0; jSub < nySub; jSub++ )
                        {
                              Camera ( x1 + hxSub*( iSub+Rnd() ), y1 + hySub*(
jSub+Rnd()), ray );
                              Color += Trace  ( Air, 1.0, ray );
                        }

                  Color /= PrimarySamples;
                  Clip ( Color );
                  c.Red   = Color.x * 255;
                  c.Green = Color.y * 255;
                  c.Blue  = Color.z * 255;
                  tga -> PutPixel ( c );
                  DrawPixel ( j, i, Color );
            }
      }
```

```
      Ticks -= * TicksPtr;
      if ( Ticks < 0l )
            Ticks = -Ticks;

      delete tga;
      getch ();
      SetMode ( 0x03 );
      printf ( "\nEnd tracing." );
      DrawTargaFile ( PicFileName );
      printf ( "\nElapsed time : %d sec. ", (int)(Ticks/18) );
   }

   void     AdaptiveDistributedRenderScene ( double HalfWidth, double HalfHeight, int nx,
int ny, int nxSub, int nySub, double Variance, char * PicFileName )
   {
      doublex, y;                    // sample point
      Ray   ray;                     // pixel ray
      doublehx = 2.0 * HalfWidth  / nx;    // pixel size
      doublehy = 2.0 * HalfHeight / ny;
      doublehxSub = hx / nxSub;
      doublehySub = hy / nySub;
      doubleDisp;                    // dispersion squared
      int   i, j;
      VectorColor;
      VectorSum;
      VectorMean;
      int   Count;
      long  Ticks = * TicksPtr;
      TargaFile * tga = new TargaFile ( PicFileName, nx, ny );
      RGB       c;

      SetMode ( 0x13 );
      SetPreviewPalette ();

      for ( i = 0, y = HalfHeight; i < ny; i++, y -= hy )
      {
            for ( j = 0, x = - HalfWidth; j < nx; j++, x += hx )
            {
                  doublex1 = x - 0.5 * hx;
                  doubley1 = y - 0.5 * hy;
                  doubled;

                  Sum   = 0;
                  Disp  = 0;
                  Count = 0;

                  do
                  {
                        for ( int iSub = 0; iSub < nxSub; iSub++ )
                              for ( int jSub = 0; jSub < nySub; jSub++ )
                              {
                                    Camera ( x1 + hxSub*( iSub+Rnd() ), y1 + hySub*(
jSub+Rnd()), ray );

                                    Color = Trace  ( Air, 1.0, ray );
                                    Sum  += Color;
                                    Disp += Color & Color;
                                    Count++;
                              }

                        Mean = Sum / Count;
                        d = ( Disp / Count - ( Mean & Mean ) ) * Count / ( Count - 1 );
                  } while ( d >= Variance && Count < 99 );

                  Clip ( Mean );
                  c.Red   = Mean.x * 255;
```

```
                c.Green = Mean.y * 255;
                c.Blue  = Mean.z * 255;
                tga -> PutPixel ( c );

                DrawPixel ( j, i, Mean );
          }
    }

    Ticks -= * TicksPtr;
    if ( Ticks < 0l )
          Ticks = -Ticks;

    delete tga;

    getch ();
    SetMode ( 0x03 );
    printf ( "\nEnd tracing." );
    DrawTargaFile ( PicFileName );
    printf ( "\nElapsed time : %d sec. ", (int)(Ticks/18) );
}
```

Для поддержки возможности вывода в несжатый 24-битовый формат TGA служит класс TargaFile, определяемый файлами Targa.h и Targa.cpp

```
//File Targa.h
#ifndef  __TARGA__
#define  __TARGA__

struct   TargaHeader
{
   char  TextSize;
   char  MapType;
   char  DataType;
   int   MapOrg;
   int   MapLength;
   char  CMapBits;
   int   XOffset;
   int   YOffset;
   int   Width;
   int   Height;
   char  DataBits;
   char  ImType;
};

#ifndef  __RGB__
#define  __RGB__
struct   RGB
{
   char  Red;
   char  Green;
   char  Blue;
};
#endif

class    TargaFile         // basic class for writing TGA image files
{
public:
   TargaFile ( char *, int, int, char * = "" );
   ~TargaFile ();

   void  PutPixel ( RGB );
private:
   TargaHeader Hdr;
   RGB         * Buffer;
   int         BufSize;
```

```
    int         pos;
    int         file;

    void        Flush ();
};

#endif
```

```
//File Targa.cpp
#include <fcntl.h>
#include <io.h>
#include <string.h>
#include <sys\stat.h>

#include "Targa.h"

TargaFile :: TargaFile ( char * name, int width, int height, char * comment )
{
    _chmod ( name, 1, 0 );              // reset file's attributes
    unlink ( name );             // remove file

    file    = open ( name, O_WRONLY | O_BINARY | O_CREAT, S_IWRITE );
    BufSize = 1000;
    Buffer  = new RGB [BufSize];
    pos     = 0;

    memset ( &Hdr, '\0', sizeof ( Hdr ) );

    Hdr.DataType = 2;
    Hdr.Width    = width;
    Hdr.Height   = height;
    Hdr.DataBits = 24;
    Hdr.ImType   = 32;

    if ( comment [0] != '\0' )
          Hdr.TextSize = strlen ( comment ) + 1;

    write ( file, &Hdr, sizeof ( Hdr ) );

    if ( Hdr.TextSize > 0 )
          write ( file, comment, Hdr.TextSize );
}

TargaFile :: ~TargaFile ()
{
    if ( pos > 0 )
          Flush ();

    delete Buffer;
    close ( file );
}

void      TargaFile :: PutPixel ( RGB color )
{
    Buffer [pos].Red   = color.Blue;     // swap red & blue colors
    Buffer [pos].Green = color.Green;
    Buffer [pos].Blue  = color.Red;

    if ( ++pos >= BufSize )             // flush buffer if full
          Flush ();
}

void      TargaFile :: Flush ()
{
    write ( file, Buffer, pos * sizeof ( RGB ) );
```

```
    pos = 0;
}
```

При этом в процесс трассировки включен предварительный показ уже просчитанной части сцены в процессе работы. Для этого используется специальная палитра из 256 цветов, в которой по 3 бита отдается под красный и зеленые цвета, а оставшиеся 2 бита - под синий. как наименее чувствительный для глаз. По завершении расчетов по построенному файлу строится специальная палитра, используемая далее для окончательного вывода изображения.

Существуют различные методы построения палитры, варьирующиеся как по временным затратам, так и по качеству получаемого изображения. Ниже рассматривается простейший метод подбора палитры, заключающийся в квантизации цветов (под каждую компоненту отводится по 5 бит - каждый цвет определяется тогда 15 битами и возможны 32 тысячи цветов) и выборе 256 наиболее часто используемых цветов.

При выборе наиболее часто встречающихся цветов под каждый возможный цвет отводится счетчик частоты его использования, затем определяются те цвета, которые действительно были использованы и при помощи стандартной процедуры быстрой сортировки qsort определяются искомые 256 цветов.

Функции, непосредственно отвечающие за рисование, содержатся в файлах Draw.h и Draw.cpp.

```
//File Draw.h
#include <dos.h>
#include "vector.h"

#ifndef  __DRAW__
#define  __DRAW__

#ifndef  __RGB__
#define  __RGB__
struct   RGB
{
    char  Red;
    char  Green;
    char  Blue;
};
#endif

void     SetMode ( int );
void     SetPalette ( RGB far * );
void     SetPreviewPalette ();
void     DrawPixel ( int, int, Vector& );
void     BuildImagePalette ( char far *, RGB * );
void     DrawImageFile ( char * );
void     DrawTargaFile ( char * );
#endif
```

```
//File Draw.cpp
#include <alloc.h>
#include <conio.h>
#include <fcntl.h>
#include <io.h>
#include <mem.h>
#include <stdio.h>

#include "Vector.h"
#include "Tracer.h"
#include "Draw.h"
#include "Targa.h"
```

```
void     SetMode ( int Mode )
{
   asm {
         mov   ax, Mode
         int   10h
   }
}

void     SetPalette ( RGB far * Palette )
{
   asm {
         push  es
         mov   ax, 1012h
         mov   bx, 0             // first color to set
         mov   cx, 256                 // # of colors
         les   dx, Palette       // ES:DX == table of color values
         int   10h
         pop   es
   }
}

void     SetPreviewPalette ()
{
   RGB   Pal [256];
   int   i;

   for ( i = 0; i < 256; i++ )
   {
         Pal [i].Red   = ( 63 *   ( i & 7 ) )         / 7;
         Pal [i].Green = ( 63 * ( ( i >> 3 ) & 7 ) ) / 7;
         Pal [i].Blue  = ( 63 * ( ( i >> 6 ) & 3 ) ) / 3;
   }

   SetPalette ( Pal );
}

void     DrawPixel ( int x, int y, Vector& Color )
{
   int   r = Color.x * 7 + 0.5;
   int   g = Color.y * 7 + 0.5;
   int   b = Color.z * 3 + 0.5;

   pokeb ( 0xA000, x + y*320, r + ( g << 3 ) + ( b << 6 ) );
}

struct   ColorData
{
   int   Hue;        // color value
   int   Freq;       // it's frequency
};

int      ColorDataComp ( const void * v1, const void * v2 )
{
   return ( (ColorData *) v2 ) -> Freq - ( (ColorData *) v1 ) -> Freq;
}

void     BuildImagePalette ( char far * ColorTrans, RGB * Palette )
{
   ColorData * ColorTable = new ColorData [8192];
   int       MinDist;
   int       d;
   unsigned    i, j;
   int       r, g, b;
   int       index;
   int       ColorsCount;
```

```
      if ( ColorTable == NULL )
      {
            printf ( "\nNo memory for ColorTable" );
            exit ( 1 );
      }
                                    // prepare used colors table ( color, frequency )
      for ( ColorsCount = 0, i = 0; i < 32768; i++ )
            if ( ColorTrans [i] > 0 && ColorsCount < 8192 )
            {
                  ColorTable [ColorsCount].Hue  = i;
                  ColorTable [ColorsCount].Freq = ColorTrans [i];
                  ColorsCount++;
            }
                                    // sort table on frequency
      qsort  ( ColorTable, ColorsCount, sizeof ( ColorData ), ColorDataComp );
      memset ( Palette, 0, 256*3 );

      for ( i = 0; i < 256 && i < ColorsCount; i++ )
      {                                         // build 5-bit values [0..31]
            Palette [i].Red   = 2 * ( ColorTable [i].Hue & 0x1F );
            Palette [i].Green = 2 * ( ( ColorTable [i].Hue >> 5 ) & 0x1F );
            Palette [i].Blue  = 2 * ( ( ColorTable [i].Hue >> 10 ) & 0x1F );
      }
                                    // find darkest color
      for ( MinDist = 1024, i = 0; i < 256; i++ )
      {
            int   d = (int)Palette [i].Red + (int)Palette [i].Green + (int)Palette
[i].Blue;

            if ( d < MinDist )
            {
                  MinDist = d;
                  index   = i;
            }
      }

      if ( index != 0 )             // and make it background
      {
            RGB   tmp = Palette [0]; // swap Palette [0] and Palette [index]

            Palette [0]     = Palette [index];
            Palette [index] = tmp;
      }

      _fmemset ( ColorTrans, 0, 32768 );   // init translation to palette color 0
      for ( i = 0; i < ColorsCount; i++ )     // for every used color find closest palette
match
      {                             // get rgb for ColorTable [i]
            r = 2 * ( ColorTable [i].Hue & 0x1F );
            g = 2 * ( ( ColorTable [i].Hue >> 5 ) & 0x1F );
            b = 2 * ( ( ColorTable [i].Hue >> 10 ) & 0x1F );
                                    // scan palette for closest match
            for ( MinDist = 1024, j = 0; j < 256; j++ )
            {
                  d = abs ( r - Palette [j].Red ) + abs ( g - Palette [j].Green ) + abs (
b - Palette [j].Blue );

                  if ( d < MinDist )
                  {
                        MinDist = d;
                        index   = j;
                  }
            }

            ColorTrans [ColorTable [i].Hue] = index;
      }
```

```
      delete ColorTable;
   }

   void     DrawImageFile ( char * PicFileName )
   {
      int      nx, ny;
      int      i, j;
      int      r, g, b;
      int      index;
      double   Max;
      RGB      Palette [256];
      Vector   LineBuffer [800];
           int       file        = open ( PicFileName, O_RDONLY | O_BINARY );
      char far * ColorTrans  = ( char far *) farmalloc ( 32768 );  // color frequencies (
translation ) table

      printf ( "\nBuilding color histogram" );

      if ( file == -1 )
           {
            printf ( "\nCannot open %s", PicFileName );
            return;
      }

      if ( ColorTrans == NULL )
      {
            printf ( "\nInsufficient memory for ColorTrans" );
            close ( file );
            return;
      }

      lseek ( file, 0l, SEEK_SET );
      read  ( file, &nx, 2 );
      read  ( file, &ny, 2 );
      read  ( file, &Max, sizeof ( Max ) );

      _fmemset ( ColorTrans, 0, 32768 );        // init frequencies

      for ( i = 0; i < ny; i++ )
      {
            read ( file, LineBuffer, nx * sizeof ( Vector ) );
            for ( j = 0; j < nx; j++ )
            {
                  Clip ( LineBuffer [j] );

                  r     = (int)( LineBuffer [j].x * 31 );
                  g     = (int)( LineBuffer [j].y * 31 );
                  b     = (int)( LineBuffer [j].z * 31 );
                  index = r | ( g << 5 ) | ( b << 10 );

                  if ( ColorTrans [index] < 255 )
                        ColorTrans [index]++;
            }
      }

      BuildImagePalette ( ColorTrans, Palette );
      SetMode ( 0x13 );
      SetPalette ( Palette );

      lseek ( file, 4l + sizeof ( Max ), SEEK_SET );
      for ( i = 0; i < ny; i++ )
      {
            read ( file, LineBuffer, nx * sizeof ( Vector ) );
            for ( j = 0; j < nx; j++ )
            {
```

```
                Clip ( LineBuffer [j] );

                r     = (int)( LineBuffer [j].x * 31 );
                g     = (int)( LineBuffer [j].y * 31 );
                b     = (int)( LineBuffer [j].z * 31 );
                index = r | ( g << 5 ) | ( b << 10 );

                pokeb ( 0xA000, j + 320*i, ColorTrans [index] );
          }
    }

    getch ();
    SetMode ( 0x03 );
    close ( file );
    farfree ( ColorTrans );
}

void     DrawTargaFile ( char * PicFileName )
{
    int      file  = open ( PicFileName, O_RDONLY | O_BINARY );

    if ( file == -1 )
    {
          printf ( "\nCannot open %s", PicFileName );
          return;
    }

    TargaHeader Hdr;
    int       r, g, b;
    int       index;
    RGB       Palette [256];
    RGB     * LineBuffer;
    char far  * ColorTrans;

    read ( file, &Hdr, sizeof ( Hdr ) );       // read header
    lseek ( file, Hdr.TextSize, SEEK_CUR );          // skip comments

    if ( Hdr.DataType != 2 )
    {
          printf ( "\nUnsupported image type." );
          close ( file );
          return;
    }
                                         // allocate space for freq/trans table
    if ( ( ColorTrans = ( char far * ) farmalloc ( 32768 ) ) == NULL )
    {
          printf ( "\nInsufficient memory for ColorTrans" );
          close ( file );
          return;
    }

    if ( ( LineBuffer = new RGB [Hdr.Width] ) == NULL )
    {
          printf ( "\nInsufficient space for Buffer" );
          farfree ( ColorTrans );
          close ( file );
          return;
    }

    _fmemset ( ColorTrans, 0, 32768 );         // init frequencies

    for ( int i = 0; i < Hdr.Height; i++ )
    {
          read ( file, LineBuffer, Hdr.Width * sizeof ( RGB ) );
          for ( int j = 0; j < Hdr.Width; j++ )
          {                              // convert to 0..31 range
```

```
                r     = LineBuffer [j].Blue  >> 3;
                g     = LineBuffer [j].Green >> 3;
                b     = LineBuffer [j].Red   >> 3;
                index = r | ( g << 5 ) | ( b << 10 );

                if ( ColorTrans [index] < 255 )
                      ColorTrans [index]++;
          }
    }

    BuildImagePalette ( ColorTrans, Palette );
    SetMode ( 0x13 );
    SetPalette ( Palette );

    lseek ( file, sizeof ( Hdr ) + Hdr.TextSize, SEEK_SET );

    for ( i = 0; i < Hdr.Height; i++ )
    {
          read ( file, LineBuffer, Hdr.Width * sizeof ( RGB ) );
          for ( int j = 0; j < Hdr.Width; j++ )
          {
                r     = LineBuffer [j].Blue  >> 3;
                g     = LineBuffer [j].Green >> 3;
                b     = LineBuffer [j].Red   >> 3;
                index = r | ( g << 5 ) | ( b << 10 );

                pokeb ( 0xA000, j + 320*i, ColorTrans [index] );
          }
    }

    close ( file );
    farfree ( ColorTrans );
    deleteLineBuffer;

    getch ();
    SetMode ( 0x03 );
}
```

Определения базовых геометрических объектов, используемых для построения простейших объектов, приведены ниже.

```
//File Geometry.h
#ifndef  __GEOMETRY__
#define  __GEOMETRY__

#include "Vector.h"
#include "Tracer.h"

#define  EPS   0.01

class    Sphere : public GObject
{
public:
   Vector      Loc;                    // center
   double      Radius;
   double      Radius2;                // squared radius

   Sphere ( Vector& c, double r ) { Loc = c; Radius = r; Radius2 = r*r; };

   virtual     int   Intersect ( Ray&, double& );
   virtual     VectorFindNormal ( Vector& );
};

class    Plane : public GObject
{
```

```
public:                                 // Plane Eq. (n,r) + D = 0
    Vectorn;                        // unit plane normal
    doubleD;                        // distance from origin

    Plane ( Vector& normal, double dist ) { n = normal; D = dist; };
    Plane ( double, double, double, double );  // ax + by + cz + d = 0

    virtual     int   Intersect ( Ray&, double& );
    virtual     VectorFindNormal ( Vector& ) { return n; };
};

class    Rect : public GObject
{
public:
    VectorLoc;
    VectorSide1, Side2;
    Vectorn;
    Vectorku, kv;
    doubleu0, v0;

    Rect ( Vector&, Vector&, Vector& );

    virtual     int   Intersect ( Ray&, double& );
    virtual     VectorFindNormal ( Vector& ) { return n; };
};

class    Triangle : public Rect
{
public:
    Triangle ( Vector& l, Vector& s1, Vector& s2 ) : Rect ( l, s1, s2 ) {};

    virtual     int   Intersect ( Ray&, double& );
};

class    Box : public GObject
{
    Vectorn [3];              // normals no sides, sides are : ( p, n ) + d = 0;
    doubled1 [3], d2 [3];           // dist, for plane eq., d1 [i] < d2 [i]
    VectorCenter;                   // center of
public:
    VectorLoc;                // origin
    Vectore1, e2, e3;         // main edges

    Box ( Vector&, Vector&, Vector&, Vector& );
    Box ( Vector&, double, double, double );

    virtual     int   Intersect ( Ray&, double& );
    virtual     Vector  FindNormal ( Vector& );

private:
    void        InitNormals ();
};

class    Cylinder : public GObject
{
    Vectore1, e2;
    doubled1, d2;                           // parameters of edges
    doubleLen;                      // length of cylinder
    doubleLen2;                     // squared length ( vector Dir squared )
    doubleRadius2;
    doubleRadius4;
public:
    VectorLoc;
    VectorDir;
    doubleRadius;
```

```
   Cylinder ( Vector&, Vector&, double );

   virtual     int   Intersect ( Ray&, double& );
   virtual     VectorFindNormal ( Vector& );
};

////////////////////////////   Lights   ////////////////////////////

class    PointLight : public LightSource
{
public:
   VectorLoc;
   doubleDistScale;

   PointLight ( Vector& l, double d = 1.0 ) : LightSource () { Loc = l; DistScale = d;
};

   virtual     doubleShadow ( Vector&, Vector& );
};

class    SphericLight : public LightSource
{
public:
   VectorLoc;
   doubleRadius;
   doubleDistScale;

   SphericLight ( Vector& l, double r, double d = 1.0 ) : LightSource () { Loc = l;
Radius = r; DistScale = d; };

   virtual     doubleShadow ( Vector&, Vector& );
};

class    SpotLight : public LightSource
{
public:
   VectorLoc;
   VectorDir;
   doubleConeAngle, EndConeAngle; // cosines of main angle and fall-off angle
   int   BeamDistribution;
   doubleDistScale;

   SpotLight ( Vector& l, Vector& d, double a, double da, int bd, double dscale = 1.0 )
: LightSource ()
   {
         Loc              = l;
         Dir              = d;
         ConeAngle        = a;
         EndConeAngle     = da;
         BeamDistribution = bd;
         DistScale        = dscale;
   };

   virtual     doubleShadow ( Vector&, Vector& );
};

//////////////////////////////////////////////////////////////////////////////

extern   doubleGeomThreshold;                // min. ray length accounted for
                                    // if ray length to intersection point is
                                    // lesser than this value, assume NO INTERSECTION
#endif
```

```
//File Geometry.cpp
#include <alloc.h>
```

```
#include <mem.h>

#include "Geometry.h"

double   GeomThreshold = 0.001;

////////////////////// Sphere methods //////////////////////

int      Sphere :: Intersect ( Ray& ray, double& t )
{
   Vectorl    = Loc - ray.Org;          // direction vector
   doubleL2OC = l & l;                  // squared distance
   doubletca  = l & ray.Dir;            // closest dist to center
   doublet2hc = Radius2 - L2OC + tca*tca;
   doublet2;

   if ( t2hc <= 0.0 )
         return 0;

   t2hc = sqrt ( t2hc );

   if ( tca < t2hc )              // we are inside
   {
         t  = tca + t2hc;
         t2 = tca - t2hc;
   }
   else                           // we are outside
   {
         t  = tca - t2hc;
         t2 = tca + t2hc;
   }

   if ( fabs ( t ) < GeomThreshold )
         t = t2;

   return t > GeomThreshold;
}

Vector   Sphere :: FindNormal ( Vector& p )
{
   return ( p - Loc ) / Radius;
}

////////////////////// Plane methods //////////////////////

Plane :: Plane ( double a, double b, double c, double d )
{
   n = Vector ( a, b, c );

   doubleNorm = !n;

   n /= Norm;
   D  = d / Norm;
}

int      Plane :: Intersect ( Ray& ray, double& t )
{
   doublevd = n & ray.Dir;

   if ( vd > -EPS && vd < EPS )
         return 0;

   t = - ( ( n & ray.Org ) + D ) / vd;

   return t > GeomThreshold;
}
```

```
///////////////////// Rect methods //////////////////////

Rect :: Rect ( Vector& l, Vector& s1, Vector& s2 )
{
   Loc   = l;
   Side1 = s1;
   Side2 = s2;
   n     = Normalize ( Side1 ^ Side2 );

   doubles11 = Side1 & Side1;
   doubles12 = Side1 & Side2;
   doubles22 = Side2 & Side2;
   doubled   = s11 * s22 - s12 * s12;  // determinant

   ku = ( Side1 * s22 - Side2 * s12 ) / d;
   kv = ( Side2 * s11 - Side1 * s12 ) / d;
   u0 = - ( Loc & ku );
   v0 = - ( Loc & kv );
}

int      Rect :: Intersect ( Ray& r, double& t )
{
   doublevd = n & r.Dir;

   if ( vd > -EPS && vd < EPS )
         return 0;

   if ( ( t = ( ( Loc - r.Org ) & n ) / vd ) < GeomThreshold )
         return 0;

   Vectorp = r.Point ( t );
   doubleu = u0 + ( p & ku );
   doublev = v0 + ( p & kv );

   return u > 0 && v > 0 && u < 1 && v < 1;
}

///////////////////// Triangle methods //////////////////////

int      Triangle :: Intersect ( Ray& r, double& t )
{
   doublevd = n & r.Dir;

   if ( vd > -EPS && vd < EPS )
         return 0;

   if ( ( t = ( ( Loc - r.Org ) & n ) / vd ) < GeomThreshold )
         return 0;

   Vectorp = r.Point ( t );
   doubleu = u0 + ( p & ku );
   doublev = v0 + ( p & kv );

   return u > 0 && v > 0 && u + v < 1;
}

/////////////////////////// Box methods ///////////////////////

Box :: Box ( Vector& l, Vector& s1, Vector& s2, Vector& s3 )
{
   Loc = l;
   e1  = s1;
   e2  = s2;
   e3  = s3;
```

```
      Center = Loc + ( e1 + e2 + e3 ) * 0.5;
      InitNormals ();
   }

   Box :: Box ( Vector& l, double a, double b, double c )
   {
      Loc = l;
      e1  = Vector ( a, 0, 0 );
      e2  = Vector ( 0, b, 0 );
      e3  = Vector ( 0, 0, c );

      Center = Loc + ( e1 + e2 + e3 ) * 0.5;
      InitNormals ();
   }

   void     Box :: InitNormals ()
   {
      n  [0] = Normalize ( e1 ^ e2 );
      d1 [0] = - ( n [0] & Loc );
      d2 [0] = - ( n [0] & ( Loc + e3 ) );

      n  [1] = Normalize ( e1 ^ e3 );
      d1 [1] = - ( n [1] & Loc );
      d2 [1] = - ( n [1] & ( Loc + e2 ) );

      n  [2] = Normalize ( e2 ^ e3 );
      d1 [2] = - ( n [2] & Loc );
      d2 [2] = - ( n [2] & ( Loc + e1 ) );

      for ( int i = 0; i < 3; i++ )
            if ( d1 [i] > d2 [i] )        // flip normals, so that d1 < d2
            {
                  d1 [i] = -d1 [i];
                  d2 [i] = -d2 [i];
                  n  [i] = -n [i];
            }
   }

   int      Box :: Intersect ( Ray& r, double& t )
   {
      doubletNear = -INFINITY;      // tNear = max t1
      doubletFar  = INFINITY;       // tFar  = min t2
      doublet1, t2;
      doublevd, vo;

      for ( int i = 0; i < 3; i++ )       // process each slab
      {
            vd = r.Dir & n [i];
            vo = r.Org & n [i];

            if ( vd > EPS )               // t1 < t2, since d1 [i] < d2 [i]
            {
                  t1 = -( vo + d2 [i] ) / vd;
                  t2 = -( vo + d1 [i] ) / vd;
            }
            else
            if ( vd < -EPS )        // t1 < t2, since d1 [i] < d2 [i]
            {
                  t1 = -( vo + d1 [i] ) / vd;
                  t2 = -( vo + d2 [i] ) / vd;
            }
            else                    // abs ( vd ) < Threshold => ray is parallel to slab
            {
                  if ( vo < d1 [i] || vo > d2 [i] )
                        return 0;
                  else
```

```
                    continue;
            }

            if ( t1 > tNear )
                  tNear = t1;

            if ( t2 < tFar )
                  if ( ( tFar = t2 ) < GeomThreshold )
                        return 0;

            if ( tNear > tFar )
                  return 0;
      }

      t = tNear;

      return t > GeomThreshold;
}

Vector   Box :: FindNormal ( Vector& p )
{
      doubleMinDist = INFINITY;
      int   index   = 0;
      doubled, Dist1, Dist2;
      Vectornormal;

      for ( int i = 0; i < 3; i++ )
      {
            d     = p & n [i];
            Dist1 = fabs ( d + d1 [i] );   // distances from point to
            Dist2 = fabs ( d + d2 [i] );   // pair of parallel planes

            if ( Dist1 < MinDist )
            {
                  MinDist = Dist1;
                  index   = i;
            }
            if ( Dist2 < MinDist )
            {
                  MinDist = Dist2;
                  index   = i;
            }
      }

      normal = n [index];                  // normal to plane, the point belongs to
      if ( ( ( p - Center ) & normal ) < 0.0 )
            normal = -normal;        // normal must point outside of center

           return normal;
}

/////////////////////////// Cylinder methods ////////////////////////////

Cylinder :: Cylinder ( Vector& l, Vector& d, double r )
{
      Loc     = l;
      Dir     = d;
      Radius  = r;
      Radius2 = r * r;
      Radius4 = Radius2 * Radius2;
      Len2    = Dir & Dir;
      Len     = ( double ) sqrt ( Len2 );

      if ( fabs ( Dir.x ) + fabs ( Dir.y ) > fabs ( Dir.z ) )
            e1 = Vector ( Dir.y, -Dir.x, 0.0 );
      else
```

```
        e1 = Vector ( 0.0, Dir.z, -Dir.y );

    e1 = Normalize ( e1 ) * Radius;
    e2 = Normalize ( Dir ^ e1 ) * Radius;
    d1 = - ( Loc & Dir );
    d2 = - ( ( Loc + Dir ) & Dir );
}

int     Cylinder :: Intersect ( Ray& r, double& t )
{
    Vectorl  = r.Org - Loc;
    doubleu0 = l & e1;
    doubleu1 = r.Dir & e1;
    doublev0 = l & e2;
    doublev1 = r.Dir & e2;
    doublel0 = l & Dir;
    doublel1 = r.Dir & Dir;
    doublea  = u1 * u1 + v1 * v1;
    doubleb  = u0 * u1 + v0 * v1;
    doublec  = u0 * u0 + v0 * v0 - Radius4;
    doubled  = b * b - a * c;

    if ( d <= 0.0 )
        return 0;

    d = sqrt ( d );

    doublet1   = ( - b - d ) / a;       // t1 < t2, since a > 0
    doublet2   = ( - b + d ) / a;
    doublelen1 = ( l0 + t1*l1 ) / Len2;
    doublelen2 = ( l0 + t2*l1 ) / Len2;
    Vectorp;
                              // now check for top/bottom intersections
    if ( l1 > EPS )
    {                         // check t1
        if ( len1 < 0.0 )     // bottom intersection
        {
            t1 = - ( ( r.Org & Dir ) + d1 ) / l1;
            p  = r.Point ( t1 ) - Loc;
            if ( ( p & p ) >= Radius2 )
                t1 = -1;
        }
        else
        if ( len1 > 1.0 )     // top intersection
        {
            t1 = - ( ( r.Org & Dir ) + d2 ) / l1;
            p  = r.Point ( t1 ) - Loc - Dir;
            if ( ( p & p ) >= Radius2 )
                t1 = -1;
        }
                              // check t2
        if ( len2 < 0 )             // bottom intersection
        {
            t2 = - ( ( r.Org & Dir ) + d1 ) / l1;
            p  = r.Point ( t2 ) - Loc;
            if ( ( p & p ) >= Radius2 )
                t2 = -1;
        }
        else
        if ( len2 > 1.0 )     // top intersection
        {
            t2 = - ( ( r.Org & Dir ) + d2 ) / l1;
            p  = r.Point ( t2 ) - Loc - Dir;
            if ( ( p & p ) >= Radius2 )
                t2 = -1;
        }
```

```
    }
    else
    if ( l1 < -EPS )
    {                               // check t1
          if ( len1 < 0 )                 // top intersection
          {
                t1 = - ( ( r.Org & Dir ) + d2 ) / l1;
                p  = r.Point ( t1 ) - Loc - Dir;
                if ( ( p & p ) >= Radius2 )
                      t1 = -1;
          }
          else
          if ( len1 > 1.0 )         // bottom intersection
          {
                t1 = - ( ( r.Org & Dir ) + d1 ) / l1;
                p  = r.Point ( t1 ) - Loc;
                if ( ( p & p ) >= Radius2 )
                      t1 = -1;
          }
                                    // check t2
          if ( len2 < 0.0 )         // top intersection
          {
                t2 = - ( ( r.Org & Dir ) + d2 ) / l1;
                p  = r.Point ( t2 ) - Loc - Dir;
                if ( ( p & p ) >= Radius2 )
                      t2 = -1;
          }
          else
          if ( len2 > 1.0 )         // bottom intersection
          {
                t2 = - ( ( r.Org & Dir ) + d1 ) / l1;
                p  = r.Point ( t2 ) - Loc;
                if ( ( p & p ) >= Radius2 )
                      t2 = -1;
          }
    }

    if ( t1 > GeomThreshold )
    {
          t = t1;
          return 1;
    }

    return ( t = t2 ) > GeomThreshold;
}

Vector   Cylinder :: FindNormal ( Vector& p )
{
    doublet = ( ( p - Loc ) & Dir )/Len2;      // parameter along Dir
    Vectorn;

    if ( t < EPS )                        // bottom
          n = - Dir / Len;
    else
    if ( t > 1.0 - EPS )                  // top
          n = Dir / Len;
    else                            // point on tube
          n = Normalize ( p - Loc - Dir * t );

         return n;
}

//////////////////////// Lights implementation ////////////////////////

double   PointLight :: Shadow ( Vector& p, Vector& l )
{
```

```
      l = Loc - p;                    // vector to light source

      doubleDist        = !l;       // distance to light source
      doubleAttenuation = DistScale / Dist;      // distance attenuation of light
      doublet;

      l /= Dist;                      // Normalize vector l

      Ray         ray ( p, l );           // shadow ray
      SurfaceData   Texture;
      GObject     * Occlude;
                                      // check all occluding objects
      Attenuation = Attenuation * Attenuation;// distance attenuation is prop. to squared
dist.

      while ( ( Occlude = Scene -> Intersect ( ray, t ) ) != NULL && Dist > t )
      {                               // adjust ray origin and get transparency koeff.
            Occlude -> FindTexture ( ray.Org = ray.Point ( t ), Texture );

            if ( Texture.Kt < Threshold ) // object is opaque
                  return 0;

            if ( ( Attenuation *= Texture.Kt ) < Threshold )
                  return 0;

            Dist -= t;
      }

      return Attenuation;
   }

   double   SpotLight :: Shadow ( Vector& p, Vector& l )
   {
      l = Loc - p;

      doubleDist        = !l;       // distance to light source
      doubleAttenuation = DistScale / Dist;      // distance attenuation of light

      l /= Dist;

      doubleld   = - ( Dir & l );

      if ( ld < EndConeAngle )
            return 0;

      double    f1 = pow ( ld, BeamDistribution );
      double    f2 = ( ld > ConeAngle ? 1.0 : ( ld - EndConeAngle ) / ( ConeAngle -
EndConeAngle ) );
      double    t;
      Ray       ray ( p, l );       // shadow ray
      SurfaceData Texture;
      GObject     * Occlude;

      Attenuation *= Attenuation * f1 * f2;

                                      // check all occluding objects
      while ( ( Occlude = Scene -> Intersect ( ray, t ) ) != NULL && Dist > t )
      {                               // adjust ray origin and get transparency koeff.
            Occlude -> FindTexture ( ray.Org = ray.Point ( t ), Texture );

            if ( Texture.Kt < Threshold ) // object is opaque
                  return 0;

            if ( ( Attenuation *= Texture.Kt ) < Threshold )
                  return 0;
```

```
                Dist -= t;
        }

        return Attenuation;
}

double   SphericLight :: Shadow ( Vector& p, Vector& l )
{
   l = Loc - p + RndVector () * Radius; // vector to random point on source

   doubleDist        = !l;        // distance to light source
   doubleAttenuation = DistScale / Dist;      // distance attenuation of light
   doublet;

   l /= Dist;                     // Normalize vector l

   Ray          ray ( p, l );            // shadow ray
   SurfaceData   Texture;
   GObject      * Occlude;
                                  // check all occluding objects
   while ( ( Occlude = Scene -> Intersect ( ray, t ) ) != NULL && Dist > t )
   {                              // adjust ray origin and get transparency koeff.
         Occlude -> FindTexture ( ray.Org = ray.Point ( t ), Texture );

         if ( Texture.Kt < Threshold ) // object is opaque
               return 0;

         if ( ( Attenuation *= Texture.Kt ) < Threshold )
               return 0;

         Dist -= t;
   }

   return Attenuation;
}
```

Следующий файл создает простейшую сцену, иллюстрирующую влияние параметров Ka, Kd, Ks и p на вид объекта.

```
//File Example1.cpp
#include "Vector.h"
#include "Tracer.h"
#include "Render.h"
#include "Geometry.h"
#include "Colors.h"

extern   unsigned  _stklen = 10240;

main ()
{
   Sphere   * s1, * s2, * s3;
   Plane      * p;
   PointLight * Light1;

   Scene = new Environment ();
   s1    = new Sphere ( Vector ( 0, 1, 5 ), 1.5 );
   s2    = new Sphere ( Vector ( -3, 0, 4 ), 1 );
   s3    = new Sphere ( Vector ( 3, 0, 4 ), 1 );
   p     = new Plane  ( Vector ( 0, 1, 0 ), 1 );

   s1 -> DefMaterial.Ka    = 0.2;
   s1 -> DefMaterial.Kd    = 0.5;
   s1 -> DefMaterial.Ks    = 0.6;
   s1 -> DefMaterial.Kr    = 0.0;
   s1 -> DefMaterial.Kt    = 0.0;
```

```
    s1 -> DefMaterial.p     = 30;
    s1 -> DefMaterial.Color = Yellow;
    s1 -> DefMaterial.Med   = Glass;

    s2 -> DefMaterial       = s1 -> DefMaterial;
    s2 -> DefMaterial.Color = Red;

    s3 -> DefMaterial       = s1 -> DefMaterial;
    s3 -> DefMaterial.Color = Blue;

    p -> DefMaterial       = s1 -> DefMaterial;
    p -> DefMaterial.Ka    = 0.1;
    p -> DefMaterial.Ks    = 0.4;
    p -> DefMaterial.Kd    = 0.5;
    p -> DefMaterial.Kr    = 0.4;
    p -> DefMaterial.Color = Blue;

    s1 -> DefMaterial.Kr = 0.3;

    Light1 = new PointLight ( Vector ( 10, 5, -10 ), 17 );

    Scene -> Add ( s1 );
    Scene -> Add ( s2 );
    Scene -> Add ( s3 );
    Scene -> Add ( p );
    Scene -> Add ( Light1 );

    Background = SkyBlue;
    SetCamera ( Vector ( 0 ), Vector ( 0, 0, 1 ), Vector ( 0, 1, 0 ) );
    RenderScene ( 1.5, 1.0, 300, 200, "EXAMPLE1.TGA" );
}
```

Для работы этого и ряда следующих примеров необходим файл Colors.h, содержащий определения ряда основных цветов.

```
//File Colors.h
#ifndef  __COLORS__
#define  __COLORS__

#ifndef  __VECTOR__
#include "vector.h"
#endif

#define  Aquamarine        Vector ( 0.439216, 0.858824, 0.576471 )
#define  Black             Vector ( 0,        0,        0        )
#define  Blue              Vector ( 0,        0,        1        )
#define  BlueViolet        Vector ( 0.623529, 0.372549, 0.623529 )
#define  Brown             Vector ( 0.647059, 0.164706, 0.164706 )
#define  CadetBlue         Vector ( 0.372549, 0.623529, 0.623529 )
#define  Coral             Vector ( 1,        0.498039, 0        )
#define  CornflowerBlue    Vector ( 0.258824, 0.258824, 0.435294 )
#define  Cyan              Vector ( 0,        1,        1        )
#define  DarkGreen         Vector ( 0.184314, 0.309804, 0.184314 )
#define  DarkOliveGreen    Vector ( 0.309804, 0.309804, 0.184314 )
#define  DarkOrchid        Vector ( 0.6,      0.196078, 0.8      )
#define  DarkSlateBlue     Vector ( 0.419608, 0.137255, 0.556863 )
#define  DarkSlateGray     Vector ( 0.184314, 0.309804, 0.309804 )
#define  DarkSlateGrey     Vector ( 0.184314, 0.309804, 0.309804 )
#define  DarkTurquoise     Vector ( 0.439216, 0.576471, 0.858824 )
#define  DimGray           Vector ( 0.329412, 0.329412, 0.329412 )
#define  DimGrey           Vector ( 0.329412, 0.329412, 0.329412 )
#define  Firebrick         Vector ( 0.9, 0.4, 0.3 )
#define  ForestGreen       Vector ( 0.137255, 0.556863, 0.137255 )
#define  Gold              Vector ( 0.8,      0.498039, 0.196078 )
#define  Goldenrod         Vector ( 0.858824, 0.858824, 0.439216 )
```

```
#define  Gray              Vector ( 0.752941, 0.752941, 0.752941 )
#define  Green             Vector ( 0,        1,        0        )
#define  GreenYellow       Vector ( 0.576471, 0.858824, 0.439216 )
#define  Grey              Vector ( 0.752941, 0.752941, 0.752941 )
#define  IndianRed         Vector ( 0.309804, 0.184314, 0.184314 )
#define  Khaki             Vector ( 0.623529, 0.623529, 0.372549 )
#define  LightBlue         Vector ( 0.74902,  0.847059, 0.847059 )
#define  LightGray         Vector ( 0.658824, 0.658824, 0.658824 )
#define  LightGrey         Vector ( 0.658824, 0.658824, 0.658824 )
#define  LightSteelBlue    Vector ( 0.560784, 0.560784, 0.737255 )
#define  LimeGreen         Vector ( 0.196078, 0.8,      0.196078 )
#define  Magenta           Vector ( 1,        0,        1        )
#define  Maroon            Vector ( 0.556863, 0.137255, 0.419608 )
#define  MediumAquamarine  Vector ( 0.196078, 0.8,      0.6      )
#define  MediumBlue        Vector ( 0.196078, 0.196078, 0.8      )
#define  MediumForestGreen Vector ( 0.419608, 0.556863, 0.137255 )
#define  MediumGoldenrod   Vector ( 0.917647, 0.917647, 0.678431 )
#define  MediumOrchid      Vector ( 0.576471, 0.439216, 0.858824 )
#define  MediumSeaGreen    Vector ( 0.258824, 0.435294, 0.258824 )
#define  MediumSlateBlue   Vector ( 0.498039, 0,        1        )
#define  MediumSpringGreen Vector ( 0.498039, 1,        0        )
#define  MediumTurquoise   Vector ( 0.439216, 0.858824, 0.858824 )
#define  MediumVioletRed   Vector ( 0.858824, 0.439216, 0.576471 )
#define  MidnightBlue      Vector ( 0.184314, 0.184314, 0.309804 )
#define  Navy              Vector ( 0.137255, 0.137255, 0.556863 )
#define  NavyBlue          Vector ( 0.137255, 0.137255, 0.556863 )
#define  Orange            Vector ( 0.8,      0.196078, 0.196078 )
#define  OrangeRed         Vector ( 0,    0,        0.498039 )
#define  Orchid            Vector ( 0.858824, 0.439216, 0.858824 )
#define  PaleGreen         Vector ( 0.560784, 0.737255, 0.560784 )
#define  Pink              Vector ( 0.737255, 0.560784, 0.560784 )
#define  Plum              Vector ( 0.917647, 0.678431, 0.917647 )
#define  Red               Vector ( 1,        0,        0        )
#define  Salmon                  Vector ( 0.435294, 0.258824, 0.258824 )
#define  SeaGreen                Vector ( 0.137255, 0.556863, 0.419608 )
#define  Sienna                  Vector ( 0.556863, 0.419608, 0.137255 )
#define  SkyBlue                 Vector ( 0.196078, 0.6,      0.8      )
#define  SlateBlue               Vector ( 0,        0.498039, 1        )
#define  SpringGreen             Vector ( 0,        1,        0.498039 )
#define  SteelBlue               Vector ( 0.137255, 0.419608, 0.556863 )
#define  Tan                     Vector ( 0.858824, 0.576471, 0.439216 )
#define  Thistle                 Vector ( 0.847059, 0.74902,  0.847059 )
#define  Turquoise               Vector ( 0.678431, 0.917647, 0.917647 )
#define  Violet                  Vector ( 0.309804, 0.184314, 0.309804 )
#define  VioletRed               Vector ( 0.8,      0.196078, 0.6      )
#define  Wheat                   Vector ( 0.847059, 0.847059, 0.74902  )
#define  White                   Vector ( 0.988235, 0.988235, 0.988235 )
#define  Yellow                  Vector ( 1,        1,        0        )
#define  YellowGreen             Vector ( 0.6,      0.8,      0.196078 )
#define  LightWood         Vector ( 0.6,      0.24,     0.1      )
#define  MedianWood        Vector ( 0.3,      0.12,     0.03     )
#define  DarkWood          Vector ( 0.05,     0.01,     0.005    )
#endif
```

Следующий пример показывает работу с простейшими объектами и отражение.

```
//File Colors.h
#include "Vector.h"
#include "Tracer.h"
#include "Render.h"
#include "Geometry.h"
#include "Colors.h"

main ()
{
```

```
    Sphere     * s [16];
    PointLight * Light1;
    int        i, j, k;

    Scene  = new Environment ();

    for ( i = k = 0; i < 4; i++ )
          for ( j = 0; j < 4; j++, k++ )
          {
                s [k] = new Sphere ( Vector ( -3 + j*2, 2.15 - i*1.45,  5 ), 0.7 );

                if ( i > 0 )
                      s [k] -> DefMaterial.Ka = 0.2;
                else
                      s [k] -> DefMaterial.Ka = j * 0.33;

                if ( i < 1 )
                      s [k] -> DefMaterial.Kd = 0;
                else
                if ( i == 1 )
                      s [k] -> DefMaterial.Kd = j * 0.33;
                else
                      s [k] -> DefMaterial.Kd = 0.4;

                if ( i < 2 )
                      s [k] -> DefMaterial.Ks = 0;
                else
                if ( i == 2 )
                      s [k] -> DefMaterial.Ks = j * 0.33;
                else
                      s [k] -> DefMaterial.Ks = 0.7;

                if ( i < 3 )
                      s [k] -> DefMaterial.p = 10;
                else
                      s [k] -> DefMaterial.p = 5 + j * 5;

                s [k] -> DefMaterial.Kt = 0;
                s [k] -> DefMaterial.Kr = 0;
                s [k] -> DefMaterial.Color = Green;
                s [k] -> DefMaterial.Med.nRefr = 1;
                s [k] -> DefMaterial.Med.Betta = 0;

                Scene -> Add ( s [k] );
          }

    Light1 = new PointLight ( Vector ( 10, 5, -10 ), 15 );

    Scene -> Add  ( Light1 );

    Background = SkyBlue;
    SetCamera ( Vector ( 0, 0, -10 ), Vector ( 0, 0, 1 ), Vector ( 0, 1, 0 ) );
    RenderScene ( 0.3, 0.2, 300, 200, "EXAMPLE2.TGA" );
}
```

Моделирование текстуры

Для придания более естественного вида сцене желательно иметь возможность менять параметры поверхности (в простейшем случае -цвет) в зависимости от положения точки на ней. Ниже будут рассмотрены различные способы достижения этого.

Существуют разные способы моделирования текстуры, но практически все они подразделяются на два основных класса:

- проективные текстуры;

•процедурные (сплошные - solid) текстуры.

Представим себе, что необходимо задать определенную текстуру (например, мрамор) какому-либо объекту.

Возможны два пути:

1. Взять изображение реальной мраморной поверхности и отобразить (спроектировать) его каким-либо образом на поверхность объекта. То есть перевести исходные трехмерные координаты точки в двумерные и использовать последние для индексации в изображение.

2. Построить некоторую функцию C(x, y, z), определяющую для каждой точки пространства (x, y, z) цвет таким образом, чтобы объект, цвет которого задается этой функцией, имел вид объекта, сделанного из мрамора.

Первый путь соответствует проективным текстурам. Он наиболее прост, однако имеет целый ряд существенных недостатков: требует большого объема памяти для хранения используемых изображений, обладает сравнительно небольшой гибкостью и к тому же сопряжен с большими сложностями в подборе способа проектирования для объектов сложной формы.

Поэтому в практических задачах, как правило, используется лишь небольшое количество стандартных вариантов проектирования: плоское (параллельное проектирование вдоль заданного направления), цилиндрическое и сферическое. Для параметрически заданных поверхностей часто в качестве проекции точки (x(u, v), y(u, v), z(u, v)) выступают значения параметров (u, v).

Второй путь не требует больших затрат памяти и одинаково хорошо работает с объектами любой (сколь угодно сложной) формы. Поскольку подобная функция обычно зависит от большого количества параметров, то, изменяя их, можно легко изменять параметры текстуры. Основным недостатком этого подхода является сложность подбора соответствующей функции.

Для использования текстур необходимы некоторые изменения в ранее введенных структурах и объектах.

Добавим в SurfaceData параметр MapCoord - результат применения проектирования к исходной точке, и введем два новых объекта -Texture и Map.

Первый из этих объектов является абстрактной моделью произвольной текстуры и содержит ссылку на содержащий его объект (object), ссылку на следующую текстуру, принадлежащую данному объекту (next), и параметры, определяющие преобразование координат перед применением текстуры (Scale, Offs). Основной метод этою класса - Apply - обеспечивает применение текстуры для изменения параметров поверхности.

Класс Map является абстрактной моделью произвольного проектирования и содержит два основных метода - Apply для проектирования точки и FindTangent, определяющий касательные векторы к координатным линиям (u = const, v = const) в заданной точке.

Некоторые изменения необходимо также внести в базовом классе GObject.

```
// Изменения в файле Tracer.h
struct   SurfaceData             // surface charactericstics at a given point
{
   doubleKa;                // ambient light coefficient
   doubleKd;                // diffuse light coefficient
   doubleKs;                // specular light coefficient
   doubleKr;                // reflected ray coefficient
   doubleKt;                // transparent light coefficient
   VectorColor;             // object's color
   MediumMed;               // medium of the object
   int   p;                 // Phong's coeff.
   Vectorn;                 // normal at a given point
   VectorMapCoord;          // mapping coordinates
};
class    Texture                              // generic texture class
{
public:
```

```
   Texture     * next;
   GObject * object;
   Vector  Offs;
   Vector  Scale;

   Texture () { next = NULL; object = NULL; Offs = 0; Scale = 1; };
   virtual     ~Texture () {};

   virtual     void  Apply ( Vector& p, SurfaceData& t ) = 0;
};

class    Map                           // generic mapping class
{
public:
   virtual     ~Map () {};             // force virtual destructor

   virtual     VectorApply ( Vector& ) = 0;   // map point
   virtual     void  FindTangent ( Vector&, Vector&, Vector& ) = 0;
};

class    GObject                       // model of an abstract geometric object
{
public:
   SurfaceData   DefMaterial;     // default material ( if no texture specified )
   Map       * Mapping;
   Texture     * Material;

   GObject () { Mapping = NULL; Material = NULL; };
   virtual     ~GObject ();

   void        FindTexture ( Vector& p, SurfaceData& t );
   void        Add ( Texture * );
   virtual     int   Intersect ( Ray&, double& ) = 0;
   virtual     VectorFindNormal ( Vector& ) = 0;
};
```

```
// Изменения в файле Tracer.cpp
GObject :: ~GObject ()
{
   if ( Mapping != NULL )
         delete Mapping;

   for ( Texture * m = Material; m != NULL; m = Material )
   {
         Material = Material -> next;
         delete m;
   }
}

void     GObject :: Add ( Texture * m )
{
   m -> next   = Material;
   m -> object = this;

   Material = m;
}

void     GObject :: FindTexture ( Vector& p, SurfaceData& t )
{
   t   = DefMaterial;
   t.n = FindNormal ( p );

   if ( Mapping != NULL )
         t.MapCoord = Mapping -> Apply ( p );
```

```
    for ( Texture * m = Material; m != NULL; m = m -> next )
          m -> Apply ( p, t );

}
```

Рассмотрение начнем с простейших сплошных текстур - клеточной (Checker) и кирпичной (Brick).

Первая разбивает все пространство на одинаковые прямоугольные клетки и каждой из них назначает один из двух цветов так, чтобы клетки, разделяющие между собой одну грань, имели разные цвета.

```
//File Checker.h
#include "Tracer.h"

class    Checker : public Texture
{
public:
    VectorColor1, Color2;

    Checker ( Vector& c1, Vector& c2 ) : Texture () { Color1 = c1; Color2 = c2; };

    virtual     void  Apply ( Vector&, SurfaceData& );
};

 //File Checker.cpp
#include "Vector.h"
#include "Tracer.h"
#include "Checker.h"

void     Checker :: Apply ( Vector& p, SurfaceData& t )
{
    Vectorr = p * Scale + Offs;
    int   ix = (int) ( r.x < 0 ? 1 - r.x : r.x );
    int   iy = (int) ( r.y < 0 ? 1 - r.y : r.y );
    int   iz = (int) ( r.z < 0 ? 1 - r.z : r.z );

    if ( ( ix + iy + iz ) & 1 )
          t.Color = Color2;
    else
          t.Color = Color1;
}
```

Текстура, задающая текстуру кирпичной стенки, оказывается, естественно, сложнее, так как кроме учета всех требуемых размеров необходимо также учитывать еще и четность/нечетность слоев по оси Oy.

```
//File Brick.h
#include "Tracer.h"

class    Brick : public Texture
{
public:
    VectorBrickSize;
    VectorMortarSize;
    VectorBrickColor, MortarColor;

    Brick ( Vector& bs, Vector& ms, Vector& bc, Vector& mc ) : Texture ()
    {
          BrickSize  = bs;  MortarSize  = ms / bs;
          BrickColor = bc;        MortarColor = mc;
    };
```

```
   virtual     void  Apply ( Vector&, SurfaceData& );
};
```

```
//File Brick.cpp
#include "Vector.h"
#include "Tracer.h"
#include "Brick.h"

void     Brick :: Apply ( Vector& p, SurfaceData& t )
{
   Vectorr = ( p * Scale + Offs ) / BrickSize;
   doublebx, by, bz;

   if ( Mod ( r.y, 1 ) <= MortarSize.y )
   {
         t.Color = MortarColor;
         return;
   }

   by = Mod ( 0.5 * r.y, 1 );
   if ( ( bx = Mod ( r.x, 1 ) ) <= MortarSize.x && by <= 0.5 )
   {
         t.Color = MortarColor;
         return;
   }

   if ( ( bx += 0.5 ) >= 1.0 )
         bx -= 1;

   if ( bx <= MortarSize.x && by > 0.5 )
   {
         t.Color = MortarColor;
         return;
   }

   if ( ( bz = Mod ( r.z, 1 ) ) <= MortarSize.z && by > 0.5 )
   {
         t.Color = MortarColor;
         return;
   }

   if ( ( bz += 0.5 ) >= 1.0 )
         bz -= 1;

   if ( bz <= MortarSize.z && by <= 0.5 )
   {
         t.Color = MortarColor;
         return;
   }

   t.Color = BrickColor;
}
```

Ниже приводится пример сцены, иллюстрирующий использование введенных текстур.

```
//File Example3.cpp
#include "Vector.h"
#include "Tracer.h"
#include "Render.h"
#include "Geometry.h"
#include "Colors.h"
#include "Brick.h"

extern   unsigned _stklen = 10240;
```

```
   main ()
   {
      PointLight * Light1, * Light2;
      Rect       * Facet1, * Facet2, * Facet3;
      Sphere     * Sphere1, * Sphere2, * Sphere3;

      Scene   = new Environment ();
      Facet1  = new Rect ( Vector ( -50, -50, -53 ), Vector ( 200, 0, 0 ), Vector ( 0, 0,
200 ) );
      Facet2  = new Rect ( Vector ( -50, -50, -53 ), Vector ( 0, 0, 200 ), Vector ( 0,
200, 0 ) );
      Facet3  = new Rect ( Vector ( -50, -50, -53 ), Vector ( 0, 200, 0 ), Vector ( 200,
0, 0 ) );
      Sphere1 = new Sphere ( Vector ( 15, 10, -30 ), 15 );
      Sphere2 = new Sphere ( Vector ( 10, -40, -5 ), 15 );
      Sphere3 = new Sphere ( Vector ( 45, -10, -20 ), 15 );
      Light1  = new PointLight ( Vector ( -20, 20, -25 ), 40 );
      Light2  = new PointLight ( Vector ( 30, -23, 15 ), 40 );

      Facet1 -> Add ( new Brick ( Vector ( 11, 6, 5 ), Vector ( 0.75 ), Firebrick, Vector
( 0.5 ) ) );
      Facet2 -> Add ( new Brick ( Vector ( 11, 6, 5 ), Vector ( 0.75 ), Firebrick, Vector
( 0.5 ) ) );
      Facet3 -> Add ( new Brick ( Vector ( 11, 6, 5 ), Vector ( 0.75 ), Firebrick, Vector
( 0.5 ) ) );

      Facet1 -> DefMaterial.Ka  = 0.25;
      Facet1 -> DefMaterial.Kt  = 0.0;
      Facet1 -> DefMaterial.Kr  = 0.0;
      Facet1 -> DefMaterial.Ks  = 0.0;
      Facet1 -> DefMaterial.Kd  = 1.0;
      Facet1 -> DefMaterial.p   = 1;
      Facet1 -> DefMaterial.Med = Air;

      Facet2 -> DefMaterial = Facet1 -> DefMaterial;
      Facet3 -> DefMaterial = Facet1 -> DefMaterial;

      Sphere1 -> DefMaterial.Ka    = 0.25;       // transparent sphere
      Sphere1 -> DefMaterial.Kd    = 0.0;
      Sphere1 -> DefMaterial.Ks    = 0.3;
      Sphere1 -> DefMaterial.Kr    = 0.3;
      Sphere1 -> DefMaterial.Kt    = 0.8;
      Sphere1 -> DefMaterial.p     = 100;
      Sphere1 -> DefMaterial.Med.nRefr = 1.35;
      Sphere1 -> DefMaterial.Med.Betta = 0;
      Sphere1 -> DefMaterial.Color = 0;

      Sphere2 -> DefMaterial.Ka    = 0.25;       // Blue sphere
      Sphere2 -> DefMaterial.Kd    = 0.4;
      Sphere2 -> DefMaterial.Ks    = 0.0;
      Sphere2 -> DefMaterial.Kr    = 0.0;
      Sphere2 -> DefMaterial.Kt    = 0.0;
      Sphere2 -> DefMaterial.p     = 3;
      Sphere2 -> DefMaterial.Med   = Glass;
      Sphere2 -> DefMaterial.Color = Blue;

      Sphere3 -> DefMaterial = Sphere1 -> DefMaterial;

      Scene -> Add ( Facet1 );
      Scene -> Add ( Facet2 );
      Scene -> Add ( Facet3 );
      Scene -> Add ( Sphere1 );
      Scene -> Add ( Sphere2 );
      Scene -> Add ( Sphere3 );
      Scene -> Add ( Light1 );
```

```
      Scene -> Add ( Light2 );

      Threshold = 0.05;
      SetCamera ( Vector ( 30, 180, 200 ), Vector ( -50, -320, -350 ), Vector ( 0, 1, 0 )
);
      RenderScene ( 150, 100, 300, 200, "sample3.tga" );
   }
```

Эти текстуры по своему действию являются цветовыми, то есть изменяющими цвет в заданной точке.

Кроме цветовых существуют также скалярные текстуры (изменяющие один из скалярных параметров, например Kt). Обычно скалярные текстуры строятся на основе цветовых, выступая как общая интенсивность

$$I = 0.229R + 0.587G + 0.114B. \quad (27)$$

Еще одним типом текстуры является текстура, изменяющая направление вектора нормали в точке. Она служит для моделирования рельефа поверхности. Аккуратное использование подобных текстур позволяет заметно усилить реалистичность получаемых изображений при сравнительно небольших вычислительных затратах. Проиллюстрируем это на примере создания бесконечной плоскости с кольцевыми волнами. Классический подход заключается в создании нового объекта - поверхности с волнами. Однако задача о пересечении такой поверхности с лучом достаточно сложна и приводит к Необходимости разрешения трансцендентных уравнений. Рассмотри другой подход. Возьмем в качестве объекта обычную плоскость, но добавим к ней текстуру, которая для создания иллюзии волн будет соответствующим образом изменять вектор нормали.

```
//File Ripples.h
#include "Tracer.h"

class    Ripples : public Texture
{
public:
   VectorCenter;
   doubleWaveLength;
   doublePhase;
   doubleAmount;

   Ripples ( Vector& c, double a, double l, double p = 0 ) : Texture ()
   {
         Center     = c;
         WaveLength = l;
         Phase      = p;
         Amount     = a;
   };

   virtual     void  Apply ( Vector&, SurfaceData& );
};
```

```
//File Ripples.cpp
#include "Tracer.h"
#include "Ripples.h"

void     Ripples :: Apply ( Vector& p, SurfaceData& t )
{
   Vectorr = p - Center;
   doublel = !r;

   if ( l > 0.0001 )
         r /= l;

   t.n += r * Amount * sin  ( 2*M_PI*l/WaveLength + Phase ) / ( 1 + l*l );
```

```
    t.n = Normalize ( t.n );
}
```

Следующая модельная сцена показывает результат применения этой текстуры.

```
// Example4.cpp
#include "Tracer.h"
#include "Geometry.h"
#include "Render.h"
#include "Colors.h"
#include "Ripples.h"

main ()
{
   Sphere     * s1, * s2, * s3;
   Plane      * p;
   PointLight * Light1;

   Scene = new Environment ();
   s1    = new Sphere ( Vector ( 0, 1, 5 ), 1 );
   s2    = new Sphere ( Vector ( -3, 0, 4 ), 1 );
   s3    = new Sphere ( Vector ( 3, 0, 4 ), 0.5 );
   p     = new Plane ( Vector ( 0, 1, 0 ), 1 );

   s1 -> DefMaterial.Ka    = 0.2;
   s1 -> DefMaterial.Kd    = 0.5;
   s1 -> DefMaterial.Ks    = 0.6;
   s1 -> DefMaterial.Kr    = 0.0;
   s1 -> DefMaterial.Kt    = 0.0;
   s1 -> DefMaterial.p     = 30;
   s1 -> DefMaterial.Color = Yellow;
   s1 -> DefMaterial.Med   = Glass;

   s2 -> DefMaterial       = s1 -> DefMaterial;
   s2 -> DefMaterial.Color = Red;

   s3 -> DefMaterial       = s1 -> DefMaterial;
   s3 -> DefMaterial.Color = Green;

   p -> DefMaterial       = s1 -> DefMaterial;
   p -> DefMaterial.Ka    = 0.1;
   p -> DefMaterial.Ks    = 0.5;
   p -> DefMaterial.Kd    = 0.4;
   p -> DefMaterial.Kr    = 0.3;
   p -> DefMaterial.Color = Blue;
   p -> Add ( new Ripples ( Vector ( 0, 0, 5 ), 2, 0.3 ) );

   Light1 = new PointLight ( Vector ( 10, 5, -10 ), 17 );

   Scene -> Add ( s1 );
   Scene -> Add ( s2 );
   Scene -> Add ( s3 );
   Scene -> Add ( p );
   Scene -> Add ( Light1 );

   Background = SkyBlue;
   SetCamera ( Vector ( 0 ), Vector ( 0, 0, 1 ), Vector ( 0, 1, 0 ) );
   RenderScene ( 1.5, 1.0, 300, 200, "SAMPLE4.TGA" );
}
```

Аналогичным путем можно получить плоские волны, рябь и ряд других эффектов.

Рассмотрим теперь достаточно широкую группу так называемых шумовых текстур.

Пусть нужно создать текстуру дерева. Известно, что дерево имеет цилиндрическую структуру (симметрию), направленную, например, вдоль оси Oz. Несложно построить функцию, которая определяет цвет, меняющийся по этому закону, например

$$C(x, y, z) = C_1 + (C_1 - C_2) f(\sqrt{x^2 + y^2}), \quad (28)$$

где C_1 и C_2 - некоторые цвета (светлых и темных колец);

$f(t)$ - некоторая неотрицательная периодическая функция, например $\frac{1}{2}(1+\sin t)$.

Ясно, что изображение, построенное при помощи подобной функции, будет симметричным и слишком правильным. На самом деле деревьев с идеальной структурой колец нет - некоторое случайное искажение присутствует практически всегда.

Для моделирования таких искажений вводится так называемая шумовая функция Noise (x, y, z). Обычно на шумовую функцию накладываются следующие требования:

1) чтобы она была непрерывной функцией,

2) принимала значения из отрезка [0, 1] и

3) вела себя в некотором смысле аналогично равномерно распределенной случайной величине. I Существует несколько способов построения подобной функции.

Простейшим из них является задание случайных значений в узлах некоторой регулярной сетки (например, в точках (i, j, к), где $i, j, k \in Z$ - целые числа) и последующей интерполяции на все остальные точки. Тем самым для отыскания значения этой функции в произвольной точке P(x, y, z) сначала определяется параллелепипед, содержащий данную точку внутри себя, затем, используя известные значения функции в вершинах этого параллелепипеда, посредством интерполяции находится и значение функции в исходной точке. В этом случае использования целочисленной решетки приходим к следующему способу задания функции:

$$Noise(x, y, z) = \sum_{i=[x]}^{[x]+1} \sum_{j=[y]}^{[y]+1} \sum_{k=[z]}^{[z]+1} \omega(|x-i|)\omega(|z-k|)a_{ijk}, \quad (29)$$

где ш(u) - одномерная весовая функция. В простейшем случае

$\omega(u) = u, u \in [0,1]$.

Однако такая трилинейная интерполяция дает не очень хорошие результаты, так как не является гладкой - на границе параллелепипедов происходит разрыв первых производных. Для достижения гладкости наложим на функцию ω (u) следующее условие:

$$\omega(0)' = \omega(1)' = 0. \quad (31)$$

Простейшим вариантом функции, удовлетворяющим этому условию, является многочлен Эрмита

$$\omega(u) = 3u^2 - 2u^3, u \in [0,1]. \quad (32)$$

Кроме шумовой функции Noise довольно часто используется также следующая функция:

$$Turbulence(p, k) = \sum_{i=1}^{k} \frac{1}{2^i} Noise(2^i p). \quad (33)$$

Далее приводится реализация этих функций, а также их векторных аналогов, с использованием описанного метода.

```
// Noise.h
#ifndef  __NOISE__
#define  __NOISE__
#include <math.h>
#include "Vector.h"

#define  NOISE_DIM   15

void     InitNoise ();
double   Noise ( const Vector& );
Vector   Noise3d ( const Vector& );
double   Turbulence ( const Vector&, int );
Vector   Turbulence3d ( const Vector&, int );

#endif
```

```
// Noise.cpp
#include <StdLib.h>
#include "Noise.h"
#include "Tracer.h"

static   doubleNoiseTable [NOISE_DIM][NOISE_DIM][NOISE_DIM];

inline   doubleSqr ( double t )
{
   return t*t;
}

inline   doubleSpline ( double t )
{
   return t * t * ( 3 - 2 * t );
}

void     InitNoise ()
{
   int   i, j, k;

   for ( i = 0; i < NOISE_DIM; i++ )
         for ( j = 0; j < NOISE_DIM; j++ )
               for ( k = 0; k < NOISE_DIM; k++ )
                     NoiseTable [i][j][k] = (double)rand () / (double)RAND_MAX;
}

double   Noise ( const Vector& p )
{
   doublesx = Mod ( p.x, NOISE_DIM );
   doublesy = Mod ( p.y, NOISE_DIM );
   doublesz = Mod ( p.z, NOISE_DIM );
   int   ix = (int) sx;
   int   iy = (int) sy;
   int   iz = (int) sz;
   int   jx = ix + 1;
   int   jy = iy + 1;
   int   jz = iz + 1;

   if ( jx >= NOISE_DIM )
         jx = 0;

   if ( jy >= NOISE_DIM )
         jy = 0;

   if ( jz >= NOISE_DIM )
         jz = 0;

   sx = Spline ( sx - ix );
   sy = Spline ( sy - iy );
   sz = Spline ( sz - iz );

   return (1-sx) * (1-sy) * (1-sz) * NoiseTable [ix][iy][iz] +
          (1-sx) * (1-sy) * sz     * NoiseTable [ix][iy][jz] +
          (1-sx) * sy     * (1-sz) * NoiseTable [ix][jy][iz] +
          (1-sx) * sy     * sz     * NoiseTable [ix][jy][jz] +
          sx     * (1-sy) * (1-sz) * NoiseTable [jx][iy][iz] +
          sx     * (1-sy) * sz     * NoiseTable [jx][iy][jz] +
          sx     * sy     * (1-sz) * NoiseTable [jx][jy][iz] +
          sx     * sy     * sz     * NoiseTable [jx][jy][jz];
}

Vector   Noise3d ( const Vector& p )
{
```

```
    Vectorres;

    doublesx = Mod ( p.x, NOISE_DIM );
    doublesy = Mod ( p.y, NOISE_DIM );
    doublesz = Mod ( p.z, NOISE_DIM );
    int   ix = (int) sx;
    int   iy = (int) sy;
    int   iz = (int) sz;
    int   jx, jy, jz;

    sx = Spline ( sx - ix );
    sy = Spline ( sy - iy );
    sz = Spline ( sz - iz );

    for ( int i = 0; i < 3; i++ )
    {
          ix = ( ix + 5 ) % NOISE_DIM;
          iy = ( iy + 5 ) % NOISE_DIM;
          iz = ( iz + 5 ) % NOISE_DIM;

          if ( ( jx = ix + 1 ) >= NOISE_DIM )
                jx = 0;

          if ( ( jy = iy + 1 ) >= NOISE_DIM )
                jy = 0;

          if ( ( jz = iz + 1 ) >= NOISE_DIM )
                jz = 0;

          res [i] = (1-sx) * (1-sy) * (1-sz) * NoiseTable [ix][iy][iz] +
                    (1-sx) * (1-sy) * sz     * NoiseTable [ix][iy][jz] +
                    (1-sx) * sy     * (1-sz) * NoiseTable [ix][jy][iz] +
                    (1-sx) * sy     * sz     * NoiseTable [ix][jy][jz] +
                    sx     * (1-sy) * (1-sz) * NoiseTable [jx][iy][iz] +
                    sx     * (1-sy) * sz     * NoiseTable [jx][iy][jz] +
                    sx     * sy     * (1-sz) * NoiseTable [jx][jy][iz] +
                    sx     * sy     * sz     * NoiseTable [jx][jy][jz];
    }

    return res;
}

double   Turbulence ( const Vector& p, int Octaves )
{
    doublek   = 1;
    doubleres = 0;
    Vectorr   = p;

    for ( int i = 0; i < Octaves; i++ )
    {
          res += Noise ( r ) * k;
          r   *= 2;
          k   *= 0.5;
    }

    return res;
}

Vector   Turbulence3d ( const Vector& p, int Octaves )
{
    doublek = 1;
    Vectorr = p;
    Vectorres ( 0 );

    for ( int i = 0; i < Octaves; i++ )
    {
```

```
            res += Noise3d ( r ) * k;
            r   *= 2;
            k   *= 0.5;
      }

      return res;
   }
```

Рассмотрим теперь реализацию текстуры дерева, полученной посредством добавления шумовой функции к формуле (28); управляя бликами в зависимости от цвета, эта текстура изменяет также и коэффициент Ks.

```
// Wood.h
#include "Tracer.h"
#include "Noise.h"

class    Wood : public Texture
{
public:
   doubleTurbScale;
   doubleRingSpacing;
   int   Squeeze;

   Wood ( double r, double t, int s = 1 ) : Texture ()
   {
         TurbScale   = t;
         RingSpacing = r;
         Squeeze     = s;
   };

   virtual     void  Apply ( Vector&, SurfaceData& );
};
```

```
// Wood.cpp
#include "Wood.h"
#include "Colors.h"

void     Wood :: Apply ( Vector& p, SurfaceData& t )
{
   doublex = p.x * Scale.x + Offs.x;
   doubley = p.y * Scale.y + Offs.y;
   doubles = pow ( SineWave ( RingSpacing*sqrt ( x*x+y*y ) + TurbScale * Turbulence (
p, 3 ) ), Squeeze );

   t.Color = ( 1 - s ) * LightWood + s * DarkWood;
   t.Ks   *= 0.3 * s + 0.7;
}
```

Параметр RingSpacing определяет расстояние между соседними кольцами, TurbScale - степень влияния шумовой функции, а Squeeze отвечает за сжатие темных колец. Следующий пример иллюстрирует использование этой текстуры.

```
// Example5.cpp
#include "Tracer.h"
#include "Geometry.h"
#include "Render.h"
#include "Colors.h"
#include "Wood.h"

main ()
{
   Box        * b = new Box ( Vector ( -1, -1, -2 ), Vector ( 2, 0, 0 ), Vector ( 0, 2,
0 ), Vector ( 0, 0, 4 ) );
   PointLight * Light1, * Light2;
```

```
    Scene = new Environment ();

    b -> DefMaterial.Ka    = 0.3;
    b -> DefMaterial.Kd    = 0.7;
    b -> DefMaterial.Ks    = 0.5;
    b -> DefMaterial.Kr    = 0.0;
    b -> DefMaterial.Kt    = 0.0;
    b -> DefMaterial.p     = 30;
    b -> DefMaterial.Color = Yellow;
    b -> DefMaterial.Med   = Glass;
    b -> Add ( new Wood ( 35, 6, 5 ) );

    Light1 = new PointLight ( Vector ( 10, 5, -10 ), 17 );
    Light2 = new PointLight ( Vector ( -10, -5, -10 ), 17 );

    Scene -> Add ( b );
    Scene -> Add ( Light1 );
    Scene -> Add ( Light2 );

    Background = SkyBlue;

    InitNoise ();
    SetCamera ( Vector ( -4, 8, -4 ), Vector ( 2, -5, 2 ), Vector ( 0, 1, 0 ) );
    RenderScene ( 1.5, 1.0, 300, 200, "SAMPLE5.TGA" );
}
```

Пример создания текстуры дерева показывает один характерный прием в моделировании текстур - строится некоторая скалярная функция, принимающая значения на [0, 1], и ее значение используется для получения нужного цвета путем интерполяции. Большим преимуществом подобного подхода является простота и адаптивность - всего лишь заменой набора цветов, используемых при интерполяции, можно сильно изменить вид материала. Вводимый далее класс ColorTable является простейшим примером кусочно-линейной интерполяции.

```
// ColorTbl.h
#ifndef  __COLOR_TABLE__
#define  __COLOR_TABLE__

#include "Vector.h"

struct   ColorTableEntry
{
    doubleta, tb;
    Vectorca, cb;
};

class    ColorTable
{
    int           ColorEntries;
    int           MaxEntries;
    ColorTableEntry * Entries;
public:

    ColorTable ( int = 10 );
    ~ColorTable () { delete Entries; };

    void  AddEntry ( double, double, Vector, Vector );
    VectorFindColor ( double );
};
#endif
```

```
// ColorTbl.cpp
```

```
#include "ColorTbl.h"

ColorTable :: ColorTable ( int size )
{
  Entries      = new ColorTableEntry [MaxEntries = size];
  ColorEntries = 0;
}

void     ColorTable :: AddEntry ( double a, double b, Vector c1, Vector c2 )
{
  if ( ColorEntries < MaxEntries - 1 )
  {
        Entries [ColorEntries].ta = a;
        Entries [ColorEntries].tb = b;
        Entries [ColorEntries].ca = c1;
        Entries [ColorEntries].cb = c2;
        ColorEntries++;
  }
}

Vector   ColorTable :: FindColor ( double value )
{
  if ( ColorEntries < 1 )
        return Vector ( 0 );

  if ( value <= Entries [0].ta )
        return Entries [0].ca;

  for ( int i = 0; i < ColorEntries; i++ )
        if ( value <= Entries [i].tb )
        {
              doublet = ( value - Entries [i].ta ) / ( Entries [i].tb - Entries [i].ta
);

              return ( 1 - t ) * Entries [i].ca + t * Entries [i].cb;
        }

  return Entries [ColorEntries-1].cb;
}
```

Аналогичным путем можно построить текстуру мрамора, возмущая текстуру, состоящую из ряда плоскостей, параллельных оси Ox, при помощи шумовой функции.

```
// Example6.cpp
#include "Tracer.h"
#include "Geometry.h"
#include "Render.h"
#include "Colors.h"
#include "ColorTbl.h"
#include "Noise.h"

class    Marble : public Texture
{
public:
  double   TurbScale;
  int      Squeeze;
  ColorTable Tbl;

  Marble ( double t = 1, int s = 1 ) : Texture (), Tbl ()
  {
        TurbScale   = t;
        Squeeze     = s;
  };
```

```
      virtual     void  Apply ( Vector&, SurfaceData& );
   };

   class    Granite : public Texture
   {
   public:
      ColorTable Tbl;

      Granite () : Texture (), Tbl () {};

      virtual     void  Apply ( Vector&, SurfaceData& );
   };

   void     Marble :: Apply ( Vector& p, SurfaceData& t )
   {
      doublex = p.x * Scale.x + Offs.x;
      doubles = pow ( SawWave ( x + TurbScale * Turbulence ( p, 4 ) ), Squeeze );

      t.Color = Tbl.FindColor ( s );
   }

   void     Granite :: Apply ( Vector& p, SurfaceData& t )
   {
      doubles = 0.5 * Turbulence ( 3 * p, 5 );

      t.Color = Tbl.FindColor ( s );
   }

   main ()
   {
      Sphere   * s1, * s2, * s3, * s4;
      PointLight * Light1;
      Marble     * m1 = new Marble;
      Marble     * m2 = new Marble;
      Granite    * g1 = new Granite;
      Granite    * g2 = new Granite;

      Scene = new Environment ();
      s1    = new Sphere ( Vector ( -2.2,  2.2, 10 ), 2 );
      s2    = new Sphere ( Vector (  2.2,  2.2, 10 ), 2 );
      s3    = new Sphere ( Vector ( -2.2, -2.2, 10 ), 2 );
      s4    = new Sphere ( Vector (  2.2, -2.2, 10 ), 2 );

      m1 -> Tbl.AddEntry ( 0.0, 0.8, Vector ( 0.9 ), Vector ( 0.5 ) );
      m1 -> Tbl.AddEntry ( 0.8, 1.0, Vector ( 0.5 ), Vector ( 0.2 ) );

      m2 -> Tbl.AddEntry ( 0.0, 0.8, Vector ( 0.8, 0.8, 0.6 ), Vector ( 0.8, 0.4, 0.4 ) );
      m2 -> Tbl.AddEntry ( 0.8, 1.0, Vector ( 0.8, 0.4, 0.4 ), Vector ( 0.8, 0.2, 0.2 ) );

      g1 -> Tbl.AddEntry ( 0.000, 0.178, Vector ( 0.831, 0.631, 0.569 ), Vector ( 0.925,
0.831, 0.714 ) );
      g1 -> Tbl.AddEntry ( 0.178, 0.356, Vector ( 0.925, 0.831, 0.714 ), Vector ( 0.871,
0.702, 0.659 ) );
      g1 -> Tbl.AddEntry ( 0.356, 0.525, Vector ( 0.871, 0.702, 0.659 ), Vector ( 0.831,
0.631, 0.569 ) );
      g1 -> Tbl.AddEntry ( 0.525, 0.729, Vector ( 0.831, 0.631, 0.569 ), Vector ( 0.937,
0.882, 0.820 ) );
      g1 -> Tbl.AddEntry ( 0.729, 1.000, Vector ( 0.937, 0.882, 0.820 ), Vector ( 0.831,
0.631, 0.569 ) );

      g2 -> Tbl.AddEntry ( 0.000, 0.241, Vector ( 0.973, 0.973, 0.976 ), Vector ( 0.973,
0.973, 0.976 ) );
      g2 -> Tbl.AddEntry ( 0.241, 0.284, Vector ( 0.973, 0.973, 0.976 ), Vector ( 0.600,
0.741, 0.608 ) );
      g2 -> Tbl.AddEntry ( 0.284, 0.336, Vector ( 0.600, 0.741, 0.608 ), Vector ( 0.820,
0.643, 0.537 ) );
```

```
      g2 -> Tbl.AddEntry ( 0.336, 0.474, Vector ( 0.820, 0.643, 0.537 ), Vector ( 0.886,
0.780, 0.714 ) );
      g2 -> Tbl.AddEntry ( 0.474, 0.810, Vector ( 0.886, 0.780, 0.714 ), Vector ( 0.996,
0.643, 0.537 ) );
      g2 -> Tbl.AddEntry ( 0.810, 0.836, Vector ( 0.996, 0.643, 0.537 ), Vector ( 0.973,
0.973, 0.976 ) );
      g2 -> Tbl.AddEntry ( 0.836, 1.000, Vector ( 0.973, 0.973, 0.976 ), Vector ( 0.973,
0.973, 0.976 ) );

      s1 -> DefMaterial.Ka    = 0.3;
      s1 -> DefMaterial.Kd    = 0.6;
      s1 -> DefMaterial.Ks    = 0.7;
      s1 -> DefMaterial.Kr    = 0.0;
      s1 -> DefMaterial.Kt    = 0.0;
      s1 -> DefMaterial.p     = 30;
      s1 -> DefMaterial.Color = Yellow;
      s1 -> DefMaterial.Med   = Glass;
      s1 -> Add ( m1 );

      s2 -> DefMaterial = s1 -> DefMaterial;
      s2 -> Add ( m2 );

      s3 -> DefMaterial = s1 -> DefMaterial;
      s3 -> Add ( g1 );

      s4 -> DefMaterial = s1 -> DefMaterial;
      s4 -> Add ( g2 );

      Light1 = new PointLight ( Vector ( 10, 5, -10 ), 17 );

      Scene -> Add ( s1 );
      Scene -> Add ( s2 );
      Scene -> Add ( s3 );
      Scene -> Add ( s4 );
      Scene -> Add ( Light1 );

      Background = SkyBlue;

      InitNoise ();
      SetCamera ( Vector ( 0 ), Vector ( 0, 0, 2 ), Vector ( 0, 1, 0 ) );
      RenderScene ( 1.5, 1.0, 300, 200, "SAMPLE6.TGA" );
   }
```

Шумовые текстуры могут изменять не только цвет, а также и вектор нормали, как текстура Bumpy.

```
// Example7.cpp
#include "Tracer.h"
#include "Geometry.h"
#include "Render.h"
#include "Colors.h"
#include "Noise.h"

class    BumpyTexture : public Texture
{
public:

   BumpyTexture () : Texture () {};

   virtual     void  Apply ( Vector&, SurfaceData& );
};

void     BumpyTexture :: Apply ( Vector& p, SurfaceData& t )
{
   t.n += 2 * ( Noise3d ( 2 * p ) - Vector ( 0.5 ) );
```

```
      t.n = Normalize ( t.n );
   }

   main ()
   {
      Sphere   * s1;
      PointLight * Light1, * Light2;

      Scene = new Environment ();
      s1    = new Sphere ( Vector ( 0, 0, 0 ), 4 );

      s1 -> DefMaterial.Ka    = 0.3;
      s1 -> DefMaterial.Kd    = 0.2;
      s1 -> DefMaterial.Ks    = 0.7;
      s1 -> DefMaterial.Kr    = 0.0;
      s1 -> DefMaterial.Kt    = 0.0;
      s1 -> DefMaterial.p     = 30;
      s1 -> DefMaterial.Color = Red;
      s1 -> DefMaterial.Med   = Glass;
      s1 -> Add ( new BumpyTexture );

      Light1 = new PointLight ( Vector ( -10, 8, -20 ), 20 );
      Light2 = new PointLight ( Vector ( 10,  8, -20 ), 20 );

      Scene -> Add ( s1 );
      Scene -> Add ( Light1 );
      Scene -> Add ( Light2 );

      Background = SkyBlue;

      InitNoise ();
      SetCamera ( Vector ( 0, 0, -7 ), Vector ( 0, 0, 1 ), Vector ( 0, 1, 0 ) );
      RenderScene ( 1.5, 1.0, 300, 200, "SAMPLE8.TGA" );
   }
```

Рассмотрим теперь реализацию проективных текстур.

Простейшим примером проекции является параллельное проектирование, реализованное в виде класса PlaneMap.

```
// PlaneMap.h
#include "Tracer.h"

class    PlaneMap : public Map
{
public:
   Vectoreu, ev;

   PlaneMap ( Vector& n, Vector& e1 )
   {
         eu = e1 - n * ( n & e1 ) / ( n & n );
         ev = n ^ e1;
   };

   virtual     VectorApply ( Vector& p ) { return Vector ( p & eu, p & ev, 0 ); };
   virtual     void  FindTangent ( Vector& p, Vector& tu, Vector& tv ){ tu = eu; tv =
ev; };
};
```

Также необходимы некоторые классы для удобного представления изображения, поэтому ниже приводится два класса: Image - абстрактная модель хранящегося изображения с возможностью произвольного доступа к пикселам и класс BMPImage, реализующий работу с несжатыми 16- или 256-цветными изображениями в формате BMP.

```
// Bmp.h
```

```
#ifndef  __BMP__
#define  __BMP__

#ifndef  __RGB__
#define  __RGB__
struct   RGB
{
   char  Red;
   char  Green;
   char  Blue;
};
#endif

class    Image
{
public:
   int   Width, Height;

   virtual     ~Image () {};

   virtual     RGB   GetPixel ( int, int ) = 0;
};

class    BMPImage : public Image
{
public:
   RGB   * Palette;
   char  * Data;

   BMPImage ( char * );
   ~BMPImage ();

   virtual     RGB   GetPixel ( int, int );
};
#endif
```

```
// Bmp.cpp
#include <mem.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>

#include "Vector.h"
#include "Bmp.h"

#define  BI_RGB0l
#define  BI_RLE8     1l
#define  BI_RLE4     2l

struct   BMPHeader
{
   int   Type;             // type of file, must be 'BM'
   long  Size;             // size of file in bytes
   int   Reserved1, Reserved2;
   long  OffBits;          // offset from this header to actual data
};

struct   BMPInfoHeader
{
   long  Size;             //
   long  Width;            // width of bitmap in pixels
   long  Height;                 // height of bitmap in pixels
   int   Planes;                 // # of planes
   int   BitCount;         // bits per pixel
   long  Compression;      // type of compression, BI_RGB - no compression
```

```
    long  SizeImage;        // size of image in bytes
    long  XPelsPerMeter;          // hor. resolution of the target device
    long  YPelsPerMeter;          // vert. resolution
    long  ClrUsed;
    long  ClrImportant;
};

struct   RGBQuad
{
    char  Blue;
    char  Green;
    char  Red;
    char  Reserved;
};

BMPImage :: BMPImage ( char * FileName )
{
    int         file = open ( FileName, O_RDONLY | O_BINARY );
    BMPHeader   Hdr;
    BMPInfoHeader     InfoHdr;
    RGBQuad           Pal [256];

    Palette = NULL;               // no data yet
    Data    = NULL;

    if ( file == -1 )       // cannot open
          return;
                            // read header data
    read ( file, &Hdr,     sizeof ( Hdr ) );
    read ( file, &InfoHdr, sizeof ( InfoHdr ) );

    int    NumColors = 1 << InfoHdr.BitCount;
    unsigned NumBytes  = (unsigned) filelength ( file ) - Hdr.OffBits;
    int    x, y;
    int    Count;
    int    Shift = ( 4 - ( InfoHdr.Width % 4 ) ) % 4;
    char   * buf   = (char *) malloc ( NumBytes );
    char   * ptr   = buf;

    if ( buf == NULL )
    {
          close ( file );
          return;
    }

    Width   = InfoHdr.Width;
    Height  = InfoHdr.Height;
    Palette = new RGB [NumColors];
    Data    = (char *) malloc ( (unsigned)Width * (unsigned)Height );

    if ( Data == NULL )
    {
          free ( buf );
          close ( file );
          return;
    }
                            // prepare palettes
    read  ( file, Pal, NumColors * sizeof ( RGBQuad ) );

    for ( int i = 0; i < NumColors; i++ )
    {
          Palette [i].Red   = Pal [i].Red;
          Palette [i].Green = Pal [i].Green;
          Palette [i].Blue  = Pal [i].Blue;
    }
                            // read raw data
```

```
   lseek ( file, Hdr.OffBits, SEEK_SET );
   read  ( file, buf, NumBytes );
   close ( file );


   memset ( Data, '\0', InfoHdr.Width*(unsigned)InfoHdr.Height );

   if ( InfoHdr.Compression == BI_RGB )
   {
         if ( InfoHdr.BitCount == 4 )  // 16-colors uncompressed
         {
               for ( y = Height - 1; y >= 0; y--, ptr += Shift )
                     for ( x = 0; x < Width; x += 2, ptr++ )
                     {
                           Data [ y * Width + x    ] = (*ptr) >> 4;
                           Data [ y * Width + x + 1] = (*ptr) & 0x0F;
                     }
         }
         else
         if ( InfoHdr.BitCount == 8 )  // 256-colors uncompressed
         {
               for ( y = Height - 1; y >= 0; y--, ptr += Shift )
                     for ( x = 0; x < Width; x++, ptr++ )
                           Data [ y * Width + x ] = *ptr;
         }
   }

   free ( buf );
}

BMPImage :: ~BMPImage ()
{
   if ( Palette != NULL )
         delete Palette;

   if ( Data != NULL )
         free ( Data );
}

RGB      BMPImage :: GetPixel ( int x, int y )
{
   return Palette [ Data [ x + y * Width ] ];
}
```

В основной цветовой проектируемой текстуре проектные координаты MapCoord преобразуются и используются для получения необходимого цвета с использованием билинейной интерполяции. При этом если координаты выходят за размер изображения, то они "заворачиваются".

```
// Map.h
#ifndef        __MAP__
#define        __MAP__

#include "Tracer.h"
#include "Bmp.h"

class    ColorMap : public Texture
{
public:
   Image * Img;

   ColorMap ( Image * i ) : Texture () { Img = i; };
   ~ColorMap () { delete Img; };

   virtual     void  Apply ( Vector&, SurfaceData& );
```

```
};

class    BumpMap : public Texture
{
public:
   Image * Img;
   double  Amount;

   BumpMap ( Image * i, double a ) : Texture () { Img = i; Amount = a; };
   ~BumpMap () { delete Img; };

   virtual     void  Apply ( Vector&, SurfaceData& );
};
#endif
```

```
// Map.cpp
#include "Tracer.h"
#include "Bmp.h"
#include "Map.h"

void     ColorMap :: Apply ( Vector& p, SurfaceData& t )
{
   doublex = Mod ( Offs.x + Scale.x * t.MapCoord.x, Img -> Width );
   doubley = Mod ( Offs.y + Scale.y * t.MapCoord.y, Img -> Height );
   int   ix = (int) x;
   int   iy = (int) y;
   int   jx = ix + 1;
   int   jy = iy + 1;

   x -= ix;
   y -= iy;

   if ( jx >= Img -> Width )            // wrap around
         jx = 0;

   if ( jy >= Img -> Height )
         jy = 0;
                                    // interpolate between corners
   RGB   c00 = Img -> GetPixel ( ix, iy );
   RGB   c01 = Img -> GetPixel ( ix, jy );
   RGB   c10 = Img -> GetPixel ( jx, iy );
   RGB   c11 = Img -> GetPixel ( jx, jy );

   t.Color.x = ((1-x)*(1-y)*c00.Red   + (1-x)*y*c01.Red   + x*(1-y)*c10.Red   +
x*y*c11.Red  ) / 255;
   t.Color.y = ((1-x)*(1-y)*c00.Green + (1-x)*y*c01.Green + x*(1-y)*c10.Green +
x*y*c11.Green) / 255;
   t.Color.z = ((1-x)*(1-y)*c00.Blue  + (1-x)*y*c01.Blue  + x*(1-y)*c10.Blue  +
x*y*c11.Blue ) / 255;
}

void     BumpMap :: Apply ( Vector& p, SurfaceData& t )
{
   doublex = Mod ( Offs.x + Scale.x * p.x, Img -> Width );
   doubley = Mod ( Offs.y + Scale.y * p.y, Img -> Height );
   int   ix = (int) x;
   int   iy = (int) y;
   int   jx = ix + 1;
   int   jy = iy + 1;

   x -= ix;
   y -= iy;

   if ( jx >= Img -> Width )            // wrap around
         jx = 0;
```

```
    if ( jy >= Img -> Height )
          jy = 0;
                                  // interpolate between corners
    RGB   c00 = Img -> GetPixel ( ix, iy );
    RGB   c01 = Img -> GetPixel ( ix, jy );
    RGB   c10 = Img -> GetPixel ( jx, iy );
    doublei00 = ( 0.229 * c00.Red + 0.587 * c00.Green + 0.114 * c00.Blue ) / 255;
    doublei01 = ( 0.229 * c01.Red + 0.587 * c01.Green + 0.114 * c01.Blue ) / 255;
    doublei10 = ( 0.229 * c10.Red + 0.587 * c10.Green + 0.114 * c10.Blue ) / 255;
    doubledu  = ( i10 - i00 ) * Amount;
    doubledv  = ( i01 - i00 ) * Amount;
    Vectortu, tv;

    if ( object -> Mapping != NULL )
          object -> Mapping -> FindTangent ( p, tu, tv );

    t.n += du * ( t.n ^ tv ) - dv * ( t.n ^ tu );
    t.n  = Normalize ( t.n );
}
```

В этих файлах содержится определение текстуры, использующей заданное изображение для отклонения нормали. При этом используется то, что соответствующая цветовая текстура порождает на поверхности объекта скалярное поле интенсивности I(u, v), где (u, v) - локальные координаты на поверхности. Если трактовать это поле как величину отклонения точки вдоль вектора нормали n, то возмущенное значение нормали оказывается очень близким к выражению:

$$n' = \frac{n + \frac{\partial I}{\partial u}[n, t_v] - \frac{\partial I}{\partial v}[n, t_u]}{\left\| n + \frac{\partial I}{\partial u}[n, t_v] - \frac{\partial I}{\partial v}[n, t_u] \right\|} \qquad (34)$$

Следующий пример иллюстрирует использование проективной текстуры, при этом используется файл 256color.bmp, входящий в стандартный комплект среды Microsoft Windows.

```
// Example8.cpp
#include "Tracer.h"
#include "Geometry.h"
#include "Render.h"
#include "Map.h"
#include "Bmp.h"
#include "Colors.h"
#include "PlaneMap.h"

main ()
{
    Box        * b      = new Box ( Vector ( 0, -2, 5 ), Vector ( 8, 0, 3 ), Vector ( -
8, 0, 3 ), Vector ( 0, -3, 0 ) );
    PointLight * Light1 = new PointLight ( Vector ( 7, 10, -10 ), 20 );
    BMPImage   * img    = new BMPImage ( "256color.bmp" );
    ColorMap   * cmap   = new ColorMap ( img );

    cmap -> Scale = 25;

    Scene  = new Environment ();

    b -> Mapping = new PlaneMap ( Vector ( 0, -1, -1 ), Vector ( 1, 0, 0 ) );
    b -> Add ( cmap );
    b -> DefMaterial.Ka    = 0.3;
    b -> DefMaterial.Kd    = 0.8;
    b -> DefMaterial.Ks    = 0.3;
    b -> DefMaterial.Kr    = 0.0;
```

```
    b -> DefMaterial.Kt    = 0.0;
    b -> DefMaterial.p     = 5;
    b -> DefMaterial.Med   = Glass;
    b -> DefMaterial.Color = 1;

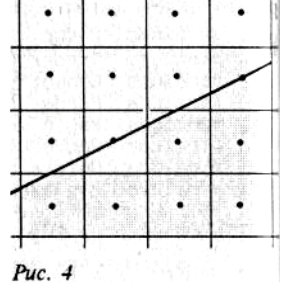
    Scene -> Add ( b );
    Scene -> Add ( Light1 );

    Background = SkyBlue;
    SetCamera ( Vector ( 0 ), Vector ( 0, 0, 1 ), Vector ( 0, 1, 0 ) );
    RenderScene ( 1.5, 1.0, 300, 200, "SAMPLE7.TGA" );
}
```

Распределенная трассировка лучей

Несложно заметить, что ряд изображений, построенных в предыдущих примерах, несет в себе заметные погрешности, наиболее заметными из которых являются "лестничные" линии и пропадающие точки.

Эти характерные явления для классической трассировки лучен носят название aliasing defects. Они связаны с тем, что каждый пиксел фактически трактуется как бесконечно малая точка на регулярной сетке, хотя на самом деле пиксел является прямоугольной областью и его цвет определяется путем суммирования по всем лучам, проходящим через эту область, т. е. является интегралом по этой области.


Рис. 4

Рассмотрим несколько примеров для объяснения этих явлений.

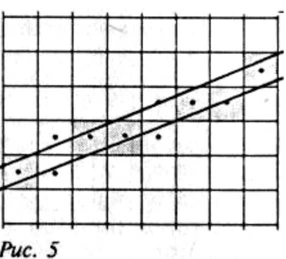
Пример 1

Пусть границей объекта является наклонная линия.

Тогда цвет пиксела однозначно определяется тем, попал ли соответствующий луч в объект или нет (см. рис. 4).

Пример 2

Рассмотрим объект с кирпичной текстурой и с достаточно тонкими линиями прослойки. Тогда, даже если большая часть лучей, проведенных через соответствующий пиксел, попадает в прослойку, возможно, что луч, выпущенный из центра пиксела, попадет мимо прослойки, и цвет пиксела ошибочно будет принят за цвет кирпича, а не прослойки. Это приводит к пропадающим точкам (рис. 5).


Рис. 5

Очевидное решение - увеличить разрешение сетки и использовать для одного пиксела не один, а несколько лучей, усредняя их значения, - способно лишь частично улучшить качество, но не в состоянии полностью избавиться от них.

Одним из наиболее распространенных и мощных средств, используемых для борьбы с этими дефектами, является так называемая распределенная трассировка лучей (Distributed Ray Tracing). При этом для вычисления соответствующего интеграла используется метод Монте-Карло.

Рассмотрим случайную точку, равномерно распределенную в области пиксела, и оттрассируем луч, проходящий через эту точку. Тогда освещенность, приносимая таким лучом, является случайной величиной и ее математическое ожидание - соответствующим интегралом. Поэтому для вычисления цвета пиксела достаточно оттрассировать несколько равномерно распределенных случайных лучей и взять среднее значение.

Существуют разные способы выбора таких точек. Наиболее распространенным является метод, основанный на "шевелении" регулярной сетки. Он заключается в том, что область пиксела разбивается на $n_1 \times n_2$ одинаковых частей и в каждой из них выбирается равномерно распределенная случайная точка, через которую проводится луч. I Этот метод позволяет полностью избавиться от aliasing-погрешностей, при этом в изображение вносится

высокочастотный шум, гораздо менее заметный для глаз, чем исходные погрешности. Реализация распределенной трассировки лучей приводится ниже.

```
// DistributedRenderScene
void    DistributedRenderScene ( double HalfWidth, double HalfHeight, int nx, int ny,
int nxSub, int nySub, char * PicFileName )
{
    double    x, y;                      // sample point
    Ray       ray;                       // pixel ray
    double    hx = 2.0 * HalfWidth  / nx;            // pixel width
    double    hy = 2.0 * HalfHeight / ny;            // pixel height
    double    hxSub = hx / nxSub;
    double    hySub = hy / nySub;
    int       i, j;
    int       PrimarySamples = nxSub * nySub;  // # of samples taken for each pixel
    Vector    Color;
    Vector    LineBuffer [800];
    long      Ticks = * TicksPtr;
    TargaFile * tga = new TargaFile ( PicFileName, nx, ny );
    RGB       c;

    SetMode ( 0x13 );
    SetPreviewPalette ();

    for ( i = 0, y = HalfHeight; i < ny; i++, y -= hy )
    {
          for ( j = 0, x = - HalfWidth; j < nx; j++, x += hx )
          {
                doublex1 = x - 0.5 * hx;
                doubley1 = y - 0.5 * hy;

                Color = 0;

                for ( int iSub = 0; iSub < nxSub; iSub++ )
                      for ( int jSub = 0; jSub < nySub; jSub++ )
                      {
                            Camera ( x1 + hxSub*( iSub+Rnd() ), y1 + hySub*(
jSub+Rnd()), ray );
                            Color += Trace  ( Air, 1.0, ray );
                      }

                Color /= PrimarySamples;
                Clip ( Color );
                c.Red   = Color.x * 255;
                c.Green = Color.y * 255;
                c.Blue  = Color.z * 255;
                tga -> PutPixel ( c );
                DrawPixel ( j, i, Color );
          }
    }

    Ticks -= * TicksPtr;
    if ( Ticks < 0l )
          Ticks = -Ticks;

    delete tga;
    getch ();
    SetMode ( 0x03 );
    printf ( "\nEnd tracing." );
    DrawTargaFile ( PicFileName );
    printf ( "\nElapsed time : %d sec. ", (int)(Ticks/18) );
}
```

Распределенная трассировка лучей позволяет также моделировать целый ряд дополнительных эффектов, таких, как неточечные источники света, нечеткое отражение, глубина резкости и другие эффекты.

Рассмотрим, каким путем это достигается.

1. Неточечные источники света

Для моделирования неточечных источников света теневые лучи трассируются в случайные точки на поверхности источника. Возможно трассирование как одного, так и нескольких теневых лучей из одной точки объекта в разные случайные точки источника света. При использовании неточечных источников света вместо резких контрастных теней возникают мягкие полутени. Реализация сферического источника света приводится ниже.

```
//SphericLight
class SphericLight : public LightSource
{
public:
 Vector Loc;
 double Radius;
 double DistScale;
 SphericLight (Vector& l, double r,double d = 1.0):
   LightSource () {Loc = l, Radius = r; DistScale = d;};
 Virtual double Shadow ( Vector&, Vector& )
}
{
 l=Loc - p +RndVector () * Radius;
 double Dist = !l;
 double Attenuation = DistScale / Dist;
 double t;
 l /=Dist;     //Normalize vector l
 Ray ray (p, l); //shadow ray
 SurfaceData Texture;
 Gobject * Occlude;
   //check all occluding objects
 while ((Occlude = Scene -> Intersect (ray,t) !=NULL && Dist>t)
{  //adjust ray origin and get transparency
 Occlude -> FindTexture(ray.Org = ray.Point( t), Texture);
 if (Texture.Kt < Treshold) return 0; //object is opaque
 if ((Attenuation *= Texture.Kt ) < Treshold ) return 0;
 Dist -=t;
 }
 return Attenuation;
}
```

2. Нечеткие отражения

Микрофасетная модель поверхности допускает нечеткое отражение, когда лучи, идущие из различных точек объекта, попадают в одну Точку поверхности и отражаются в одном и том же направлении.

Поэтому вместо одного отраженного (преломленного) луча можно выпустить несколько случайных лучей и определить приносимую ими энергию с учетом их весовых коэффициентов (23), зависящих от их направления. Вместо использования равномерно распределенных лучей удобнее использовать вес луча как плотность вероятности.

Тогда большинство выпущенных лучей попадут в направления, дающие наибольший вклад. Приносимые ими значения просто усредняя уже без весовых коэффициентов.

3. Глубина резкости

Рассматриваемая до сих пор модель камеры является идеальной в том смысле, что все объекты в ней всегда находятся в фокусе. На самом деле реальные оптические приборы не могут одновременно удерживать в фокусе всю сцену (глаз не является исключением). При этом

объекты, не попавшие в фокус, оказываются размытыми. В ряде случаев желательно иметь возможность воспроизводить этот эффект.

Рассмотрим механизм возникновения этого явления. В используемой ранее точечной камере для любой точки P экрана существует единственный луч, освещающий точку и проходящий через отверстие O. В реальной камере произвольная точка P освещается бесконечным количеством различных лучей, проходящих через линзу (рис. 6).

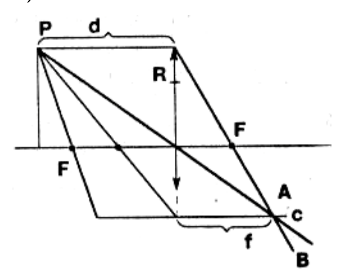


Рис. 6

Объект A, лежащий на пересечении всех этих лучей, всегда находится в фокусе, так как все лучи, идущие в линзу, попадают в одну и ту же точку экрана P. С другой стороны, объекты B и C находятся не в фокусе, так как существует луч, идущий от C и попадающий в ту же точку экрана, что и луч от объекта B. Если ранее определение луча, соответствующего точке P экрана, было тривиальным - луч проводился через точки P и O, то теперь мы должны поступить следующим образом: выберем случайную точку R на линзе и проведем через нее луч. В силу законов оптики в точке R этот луч. преломится и пройдет через точку A. Если точка P имеет координаты (-d, y), точка R - $(0, y_0)$, а точка A - $\left(f, -y\frac{f}{d}\right)$, где величины f и d связаны соотношением

$$\frac{1}{F} = \frac{1}{f} + \frac{1}{d} \qquad (35)$$

(здесь F - фокусное расстояние линзы), то направляющим вектором для луча, выходящего из точки R, будет вектор

$$v\left(f, -y\frac{f}{d} - y_0\right).$$

Основным недостатком метода распределенной трассировки лучей является заметное увеличение объема вычислений. Поэтому на практике применяются адаптивные методы, ограничивающие количество выпускаемых первичных лучей. Все эти методы основаны на статистическом анализе значений, принесенных по уже оттрассированным лучам для данного пиксела (а иногда и для соседних).

Пусть выпущено n первичных лучей, каждый из которых дает значение соответственно $c_1, c_2, \ldots, c_n$. Тогда по этим значениям можно найти несмещенные оценки математического ожидания $\hat{c}$ и дисперсии $\hat{\sigma}^2$:

$$\hat{c} = \frac{1}{n}\sum_{i=1}^{n} c_i, \hat{\sigma}^2 = \frac{n}{n-1}\left(\frac{1}{n}\sum_{i=1}^{n} c_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} c_i\right)^2\right). \qquad (36)$$

В качестве критерия точности проще использовать оценку дисперсии. Так как все переменные $c_1, c_2, \ldots, c_n$ одинаково распределены, по центральной предельной теореме случайная величина $\hat{c}$ стремится к случайной величине с нормальным законом распределения

$$N\left(c, \left(\frac{\sigma}{\sqrt{n}}\right)^2\right),$$

где c и σ - истинные значения математического ожидания и дисперсии. Отсюда следует, что, как только величина $\frac{\sigma}{\sqrt{n}}$ станет достаточно малой, можно считать, что требуемая точность достигнута с заданной вероятностью.

$$P\left\{\left|c-\hat{c}\right|<\frac{\varepsilon\sigma}{\sqrt{n}}\right\}=P\left\{\left|\sqrt{n}\,\frac{c-\hat{c}}{\sigma}\right|<\varepsilon\right\}=\frac{1}{\sqrt{2\pi}}\int_{-\varepsilon}^{\varepsilon}e^{-\frac{u^2}{2}}du\,. \qquad (37)$$

Приведенная ниже процедура AdaptiveDistributedRenderScene осуществляет распределенную трассировку сцены с использованием изложенного критерия точности.

```
// AdaptiveDistrubutedRenderScene
void AdaptiveDistributedRenderScene ( double HalfWidth, double HalfHeight, int nx, int
ny, int nxSub, int nySub, double
Variance, char * PicFileName )
{
double x, y;   // sample point
Ray ray;       // pixel ray
double hx = 2.0 * HalfWidth / nx; // pixel size
double hy = 2.0 * HalfHeight / ny;
double hxSub = hx / nxSub;
double hySub = hy / nySub;
double Disp;   // dispersion squared
int i, j;
Vector Color;
Vector Sum;
Vector Mean;
int Count;
long Ticks = * TicksPtr;
TargaFile * tga = new TargaFile ( PicFileName, nx, ny );
RGB c;
SetMode ( 0x13 );
SetPreviewPalette ();
for ( i = 0. y = HalfHeight; i < ny; i++, y -= hy )
{
 for ( j = 0, x = - HalfWidth; j < nx; j++, x += hx )
 {
  double x1 = x - 0.5 * hx:
  double y1 = y - 0.5 * hy; -C
  double d;
  Sum = 0; Disp = 0; Count = 0;
  do
  {
   for ( int iSub = 0; iSub < nxSub; iSub++ )
    for ( int jSub = 0; jSub < nySub; jSub++ ) {
     Camera (x1+hxSub-(iSub+Rnd()), y1+hySub*(jSub+Rnd()), ray);
     Color = Trace ( Air, 1.0. ray ):
     Sum += Color;
     Disp += Color & Color;
     Count++;
    }
   Mean = Sum / Count;
   d = (Disp / Count - (Mean & Mean)) * Count / (Count - 1);
  } while ( d / Count >= Variance && Count < 99 );
  Clip ( Mean );
  c.Red = Mean.x * 255;
  c.Green = Mean.y * 255;
  c.Blue = Mean.z * 255;
  tga -> PutPixel ( c ):
  DrawPixel ( i, i, Mean );
 }
}
Ticks -= * TicksPtr;
if ( Ticks < 0l ) Ticks = -Ticks;
delete tga;
getch ();
SetMode ( 0x03 );
printf ( "\nEnd tracing." );
```

```
DrawTargaFile ( PicFileName );
printf C "\nElapsed time : %d sec. ", (int)(Ticks/18) );
}
```

Методы оптимизации

Метод трассировки лучей отличает высокий объем вычислений, причем по оценкам до 95% работы в сложных сценах уходит на проверки пересечения луча с объектами сцены. Для реальных сцен, содержащих многие тысячи и десятки тысяч объектов, простой перебор для определения ближайшей точки пересечения луча с объектами сцены просто неприемлем.

Существуют методы, позволяющие заметно сократить для каждого луча количество проверяемых объектов. Ниже дается краткий обзор этих методов.

Предположим, что в рассматриваемой сцене имеется сложный объект. Поместим его внутрь достаточно простой выпуклой фигуры (например, сферы). Ясно, что если луч не пересекает эту фигуру, то он не сможет пересечь и охватываемый ею сложный объект. Более того, построенную вспомогательную фигуру пересечет лишь небольшая часть рассматриваемых лучей и только эти лучи нужно проверить на наличие пересечения с взятым объектом сцены. Это означает, что подобная двухэтажная проверка оказывается заметно лучше, чем непосредственная проверка пересечения лучей со сложным объектом:

1) поиск точек пересечения со сферой прост и алгоритмически легко реализуем;

2) количество лучей, которые нужно проверить на пересечение с исходным объектом, становится значительно меньше исходного.

Тем самым выигрыш очевиден.

Можно пойти дальше - описать простую фигуру сразу вокруг нескольких объектов. Тогда если луч не пересекает ограничивающую фигуру, то он не сможет пересечь ни одного из содержащихся внутри объектов.

Следующий шаг - оптимизация случая, когда луч все-таки пересекает ограничивающую фигуру.

Разобьем содержащиеся внутри нее объекты на группы и вокруг каждой из них опишем новую фигуру, содержащуюся в исходной. Это позволит вместо непосредственной проверки содержащихся внутри объектов проверить сначала ограничивающие их фигуры.

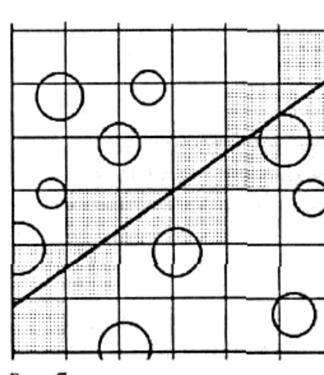
Продолжая подобный процесс, приходим к следующей организации сцены: вся сцена содержится внутри некоторой фигуры B_1, все объекты разбиты на несколько групп, и вокруг каждой из них описано по ограничивающей простейшей фигуре $B_{11},\ldots,B_{1n}$, все объекты, содержащиеся внутри каждой из этих фигур, снова разбиты на группы, вокруг каждой из которых описано по ограничивающей фигуре, и так далее.

В результате приходим к древовидной организации сцены.

```
// Bounding Volume Hierarchy
class BoundingVolume
{
public:
 BoundingVolume * child;
 BoundingVolume * next;
 GObject * Obj;
 virtual int Check ( Ray& ) = 0;
 GObject * Intersect ( Ray&, double& );
};
GObject * BoundingVolume :: Intersect ( Ray& ray, double& t )
{
 if ( ! Check ( ray ) ) return NULL;
 GObject * Found = NULL;
 GObject * tmp;
 BoundingVolume * vol = child;
 double t1;
 for ( t = INFINITY; vol != NULL; vol = vol -> next )
```

```
    if ( ( tmp = vol -> Intersect ( ray. t1 ) ) != NULL )
     if ( t1 < t ) {
      Found = tmp;
      t = t1;
     }
   if (Obj != NULL )
    if ( Obj -> Intersect ( ray. t1 ) )
     if ( t1 < t ) <
      Found = Obj;
      t = t1;
  }
  return Found;
  }
```

В качестве ограничивающих тел обычно выбирают сферы или пересечение полупространств.


Рис. 7

Количество проверок на пересечения для данного метода составляет O(log n), где n - общее количество объектов в сцене.

Основным недостатком метода дерева ограничивающих фигур является необходимость построения структуры ограничивающих фигур, что зачастую представляет собой значительные сложности.

В отличие от метода дерева ограничивающих объемов, разбивающего объекты на группы, метод разбиения пространства проводит разбиение самого пространства.

Рассмотрим простейший вариант этого метода - метод равномерного разбиения пространства.

Опишем вокруг всей сцены прямоугольный параллелепипед и разобьем его на n_1 x n_2 x n_3 равных частей. Для каждой из этих частейсоставим список всех объектов, границы которых пересекают данный блок.

Процесс нахождения ближайшего пересечения луча с объектами сцены начинается с определения части, содержащей начало луча, и двух списков - списка уже проверенных объектов и списка найденных точек пересечения, отсортированный по расстоянию до начала луча. Проверим все объекты из списка текущего блока, которые еще не были проверены. После этого добавим проверенные объекты в список уже проверенных, а все найденные пересечения - в список точек пересечения. Если ближайшая точка пересечения из списка найденных содержится в текущем блоке, то она возвращается как искомое пересечение. В случае, если в текущем блоке нет пересечений, находится следующий блок, через который проходит луч, и для него повторяется та же процедура. В случае, если следующего блока нет (луч выходит из сцены), возвращается отсутствие пересечений.

К несомненным преимуществам относится простота разбиения, возможность использования алгоритма Брезенхейма для нахождения следующего блока и направленный перебор вдоль луча, когда найденное пересечение гарантированно является ближайшим. При соответствующем выборе шагов разбиения среднее количество проверяемых объектов практически не зависит от общего количества объектов в сцене.

МЕТОД ИЗЛУЧАТЕЛЬНОСТИ

Основными недостатками метода трассировки лучей являются неэффективность работы с диффузными поверхностями и то, что определение освещенности поверхностей проводится параллельно с построением изображения и зависит от положения наблюдателя так, что любое изменение положения наблюдателя ведет к полному пересчету всей сцены.

Метод излучательности устраняет эти недостатки, обеспечивая одновременно и высокую точность при работе с диффузными объектами, и отдельное вычисление глобальной освещенности независимо от положения наблюдателя.

В основе метода излучательности лежит закон сохранения энергии в замкнутой системе. Все объекты разбиваются на фрагменты и для этих фрагментов составляются уравнения баланса энергии.

Пусть все объекты являются чисто диффузными, т. е. отражают(рассеивают) свет равномерно по всем направлениям. Разобьем всю сцену на n фрагментов и пусть

B_i - энергия, отбрасываемая i-м фрагментом сцены;

E_i - собственная излучательность фрагмента;

F_{ij} - доля энергии j-го фрагмента, попадающая на i-й фрагмент (коэффициенты формы);

ρ_i - коэффициент отражения.

Тогда уравнения баланса энергии имеют вид:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j, i = 1,\dots,n. \qquad (1)$$

Эти соотношения можно переписать в следующей форме:

$$\sum_{j=1}^{n} \left(\delta_{ij} - \rho_i F_{ij}\right) B_j = E_i, i = 1,2,\dots,n. \qquad (2)$$

или, в матричном виде,

$$\left(I - \rho F\right) B = E, \qquad (3)$$

где I - единичная матрица.

В результате получаем систему линейных алгебраических уравнений. Из закона сохранения энергии следует, что

$$\sum_{j=1}^{n} F_{ij} < 1$$

для всех i. Тем самым эта линейная система обладает так называемым диагональным преобладанием, что позволяет использовать для ее решения эффективные итерационные методы (типа Гаусса-Зейделя), дающие за небольшое число итераций вполне удовлетворительное решение.

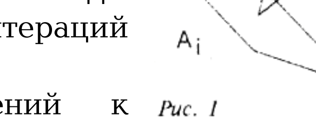


Рис. 1

Соответствующую последовательность приближений к решению можно построить, например, по следующим формулам:

$$B_i^{(0)} = E_i, \qquad (4)$$

$$B_i^{(k+1)} = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j^{(k)}. \qquad (5)$$

Итерационный процесс прекращается, когда разница между двумя последовательными приближениями оказывается меньше заданной точности.

Для определения цвета фрагмента соответствующие линейные системы записываются для каждой из трех основных цветовых составляющих, причем коэффициенты формы определяются только геометрией сцены и от цвета не зависят.

Обычно после определения освещенности каждого фрагмента производится билинейная интерполяция освещенности по всем объектам, дающая плавное естественное освещение.

После выбора точки наблюдения объекты сцены проектируются на картинную плоскость и строится изображение.

Наиболее трудоемким шагом метода излучательности является вычисление коэффициентов формы, хранящих в себе информацию о геометрии сцены. Рассмотрим этот процесс подробнее.

Выберем фрагменты A_i и A_j и элементарные участки dA_i и dA_j на них с нормалями соответственно n_i и n_j (рис. 1). Тогда доля энергии элемента dA_j, попадающей на элемент dA_i, будет равна

$$F(dA_i, dA_j) = \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2}, \qquad (6)$$

где r - расстояние между элементами dA_i и dA_j;
(φ_i и φ_j - углы между нормалями к ним и соединяющим их отрезком.

$$F_{ij} = F(A_i, A_j) = \frac{1}{A_i}\int_{A_i}\int_{A_j} \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2} dA_j dA_i. \qquad (7)$$

Легко видеть, что

$$A_i F_{ij} = A_j F_{ji}. \qquad (8)$$

В результате двойного интегрирования получаем следующее соотношение:

Интегральная формула не учитывает объектов, закрывающих часть одного фрагмента от другого. Для их учета в подынтегральное выражение нужно добавить еще один множитель - функцию HID_{ij}, принимающую значения из отрезка [0, 1] и характеризующую степень видимости точки фрагмента A_i из точки фрагмента A_j:

$$F_{ij} = \frac{1}{A_i}\int_{A_i}\int_{A_j} \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2} HID_{ij} dA_j dA_i. \qquad (9)$$

Точное вычисление этого интеграла, как правило, представляет значительные трудности. Поэтому для отыскания коэффициентов формы используется ряд достаточно эффективных приближенных методов. Наиболее распространенным является метод полукуба. Коротко опишем его.

Будем считать, что расстояние между фрагментами по сравнению с их размерами достаточно велико. Тогда, выбрав в качестве точки на фрагменте A_i его центр, можно записать приближенно, что

$$F_{ij} = \int_{A_j} \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2} HID_{ij} dA_j. \qquad (10)$$

Построим воображаемый куб таким образом, чтобы его центр совпал с центром фрагмента, а за направление оси Oz (в системе координат куба) возьмем нормаль к фрагменту в его центре (рис. 2).

Разобьем часть поверхности куба, лежащую в плоскости $z > 0$, на квадратные пикселы и спроектируем все пространство на 5 граней получившегося полукуба. Для каждого пиксела полукуба определяется ближайший проектируемый на него фрагмент (например, с использованием z-буфера), после чего вычисляется вклад в i-ю строку матрицы коэффициентов формы каждого пиксела полукуба.

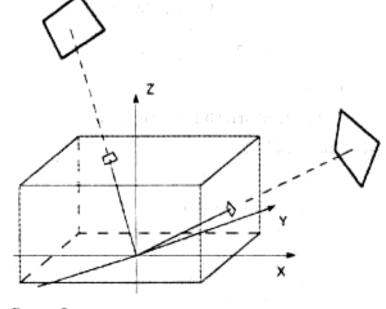


Рис. 2

Если пиксел лежит на верхней грани, то его вклад в коэффициент формы равен

$$\frac{\Delta A}{\pi(x^2 + y^2 + 1)^2}, \qquad (11)$$

где ΔA - площадь соответствующего пиксела, а для пиксела на боковой стороне -

$$\frac{z\Delta A}{\pi(x^2+y^2+1)^2}. \qquad (12)$$

Таким образом, коэффициенты формы от A_i определяются сразу ко всем остальным фрагментам.

Замечание

К сожалению, методу полукуба присущи некоторые недостатки, в частности он плохо работает с близкорасположенными гранями.

В ряде случаев выражение (9) можно заметно упростить.

Рассмотрим случай, когда грани A_i и A_j являются плоскими многоугольниками и для них функция $HID_{ij} = 1$. Перепишем формулу (8) в векторном виде:

$$F_{ij} = \frac{1}{\pi A_i} \int_{A_i} \int_{A_j} \frac{(n_i, r_j - r_i)(n_j, r_i - r_j)}{\left\| r_i - r_j \right\|^4} dA_i dA_j. \qquad (13)$$

Дважды применив к этому интегралу формулу Стокса, его можно записать в виде двойного контурного интеграла

$$F_{ij} = \frac{1}{4\pi A_i} \int_{\partial A_i} \int_{\partial A_j} \log \left\| r_i - r_j \right\|^2 dl_i dl_j. \qquad (14)$$

Так как обе грани являются многоугольниками, то этот интеграл распадается на двойную сумму интегралов по ребрам граней.

Для сокращения объема вычислений иногда используется метод прогрессивной излучательности, в котором также строится последовательность приближенных решений, но на этот раз коэффициенты формы вычисляются не все сразу, а только по мере необходимости.

Начальное приближение итерационной последовательности определяется формулой (4).

Для получения очередного приближения выбирается фрагмент A_i с наибольшей излучательностью, вычисляются коэффициенты формы от него ко всем остальным фрагментам и затем излучательности всех этих фрагментов корректируются по формуле

$$B_j^{(k+1)} = E_j + \rho_j F_{ij} B_j^{(k)},\ j = 1,\dots,n \qquad (15)$$

После того как искомые значения B_i, $i = 1,\dots,n$ найдены, строится изображение сцены с использованием какого-либо метода удаления невидимых поверхностей, например z-буфера. При этом значения излучательности обычно интерполируются для получения плавного перехода освещенности между отдельными фрагментами.

ГРАФИЧЕСКИЙ ПАКЕТ 3D STUDIO

Существует целый ряд пакетов трехмерной графики, предназначенных для создания высококачественных изображений трехмерных сцен и анимации. Подобные пакеты основаны на использовании соответствующих методов построения реалистических изображений, удаления невидимых частей, геометрического моделирования. При этом центр тяжести переносится с методов и самого процесса создания реалистических изображений на вопросы геометрического моделирования: пользователю достаточно лишь задать геометрию сцены, используемые материалы, источники света и камеры - и пакет сам построит соответствующее изображение. От пользователя при этом не требуется практически никаких специальных знаний по методам создания изображений - все необходимое уже заложено в пакете.

При помощи графических пакетов можно легко создавать изображения сцен и рекламные ролики, превращать в наглядные изображения понятные лишь специалистам чертежи.

Подобные пакеты обычно требуют достаточно больших вычислительных ресурсов, поэтому большинство из них реализовано на достаточно мощных рабочих станциях.

Однако существуют пакеты, рассчитанные на машины типа IBM AT 386. Одним из самых популярных и удобных пакетов трехмерной графики для IBM-совместимых компьютеров является пакет 3D Studio фирмы AutoDesc Inc.

Пакет 3D Studio обладает широкими возможностями по созданию высококачественных фотореалистических изображений и анимаций. При помощи этого пакета можно легко создавать на персональном компьютере то, что раньше требовало привлечения мощных рабочих станций. Поэтому совсем неудивительно, что большая часть современной телерекламы на российских студиях создается именно посредством 3D Studio.

На данный момент наиболее распространенной является третья версия пакета 3D Studio. Основными отличиями этой версии от предыдущей являются: заметное ускорение процесса построения изображения (рендеринга), добавление новой модели рендеринга - металлического, применение для отслеживания теней метода трассировки лучей. Кроме того, третья версия пакета включает в себя поддержку DPMI (дающую возможность работать в Windows и OS/2 и осуществлять там фоновый рендеринг) и поддержку формата JPEG. Одной из наиболее привлекательных возможностей третьей версии является так называемый сетевой рендеринг, когда при наличии только одной зарегистрированной копии программы расчет анимационного фильма может проводиться одновременно на нескольких машинах, объединенных в локальную сеть.

Для работы с пакетом 3D Studio требуются 100-% совместимый компьютер с процессором не ниже 386, сопроцессор (если основной процессор 386 или 486SX), 8 Мбайт оперативной памяти (для не очень сложных сцен достаточно 4 Мбайт), 20 Мбайт на жестком диске, SVGA-карта с поддержкой режима 640*480*256 цветов и Microsoft-совместимая мышь.

При наличии CD-дисковода, можно воспользоваться предлагаемыми фирмой Autodesk компакт-дисками World Creating ToolKit (содержит большое количество разнообразных готовых трехмерных объектов) и Texture Universe (содержит свыше 400 различных текстур и материалов).

Пакет защищен от копирования электронным ключом.

В комплект поставки третьей версии входят 4 книги руководства пользователя ("Autodesk 3D Studio Reference Manual", "Tutorials", "Installation Guide", "Advanced User's Guide"), 8 дискет, электронный ключ и World Creating Toolkit CD.

Структурно пакет состоит из следующих модулей:

- 2dShaper - модуль двумерного моделирования - позволяет строить плоские формы (shapes), используемые для создания плоских и трехмерных объектов, и траектории (paths) для перемещения объектов;
- 3dLofter - модуль создания трехмерных объектов путем "проноса" ("вытягивания") (lofting) плоской фигуры вдоль заданной траектории; при этом проносимая плоская фигура может подвергаться определенным деформациям, что позволяет создавать достаточно сложные объекты;
- 3dEditor - модуль создания и редактирования трехмерных объектов - позволяет как создавать простейшие объекты (параллелепипеды, сферы, цилиндры и др.), так и редактировать уже существующие; этот модуль предоставляет возможность строить новые объекты посредством логических операций (объединения, пересечения, разности) над уже имеющимися, позволяет создавать источники света и камеры и, кроме того, отвечает за назначение текстуры различным объектам и их частям;
- KeyFramer - анимационный модуль - позволяет строить анимации путем задания так называемых ключевых кадров (key frames) на основе сцены, созданной в модуле 3dEditor; в этих ключевых кадрах задаются основные преобразования объектов, которые далее интерполируются на все промежуточные кадры;
- Materials Editor - редактор материалов - позволяет просматривать, редактировать и создавать материалы для их последующего использования в сцене.

Если возможностей, предоставляемых этими модулями, оказывается недостаточно, то можно воспользоваться внешними модулями -так называемыми IPAS-процессами. Каждый такой модуль представляет собой процедуру на языке С, предназначенную для работы с 3D Studio. Третья версия пакета поддерживает 6 различных типов процессов для обработки изображений, текстур, объектов. Эти модули позволяют добиваться значительных результатов при построении сложных анимаций (например, при рассыпании объекта на мелкие и мельчайшие фрагменты).

Сцена в пакете 3D Studio состоит из объектов, источников света и камер.

Каждый объект представлен в виде наборов треугольных граней (mesh object). Для удаления невидимых граней используется метод z-буфера. Каждой грани назначается материал, обеспечивающий требуемый вид объекта. В состав пакета входит большая библиотека стандартных материалов; кроме того, пользователь может легко изменять старые материалы и создавать новые. Поддерживается 4 модели рендеринга - плоская, Гуро, Фонга и металлическая.

К пакету прилагается полная документация, подробнейшим образом описывающая все возможности пакета, его особенности, команды меню и многое другое. В комплект документации входит учебник по пакету, позволяющий легко освоить основные возможности этого пакета и приступить к созданию собственных изображений.

Осенью 1994 года в продажу поступила четвертая версия пакета 3D Studio, включающая в себя целый ряд новых возможностей. Укажем лишь некоторые из них: ускоренный предварительный просмотр процесса анимации (preview), обратная кинематика (inverse cinematics), позволяющая наглядно оперировать с иерархическими объектами, язык управления анимацией (KeyFramer Script language), сплайновые поверхности, вписывание изображения в готовую фотографию и другие.

Перейдем к описанию непосредственной работы с пакетом 3D Studio.

Для того, чтобы показать его возможности, рассмотрим следующие задачи: создание изображения надписи над поверхностью стола и ее вращение и создание тела вращения - чашки на блюдце.

Соответствующее изложение построим как бы в двух слоях. Часть материала будет написана в виде прямого обращения к пользователю этого пакета, где ему будет предложено совершить вместе с авторами некоторую осмысленную совокупность последовательных

шагов. Второй слой будет содержать пояснения к событиям, которые будут при этом разворачиваться на экране.

Для начала работы запустите 3D Studio и перейдите в 2dShaper нажатием клавиши F1.

Во всех модулях 3D Studio (кроме Materials Editor) экран устроен практически одинаковым образом (рис. 1).

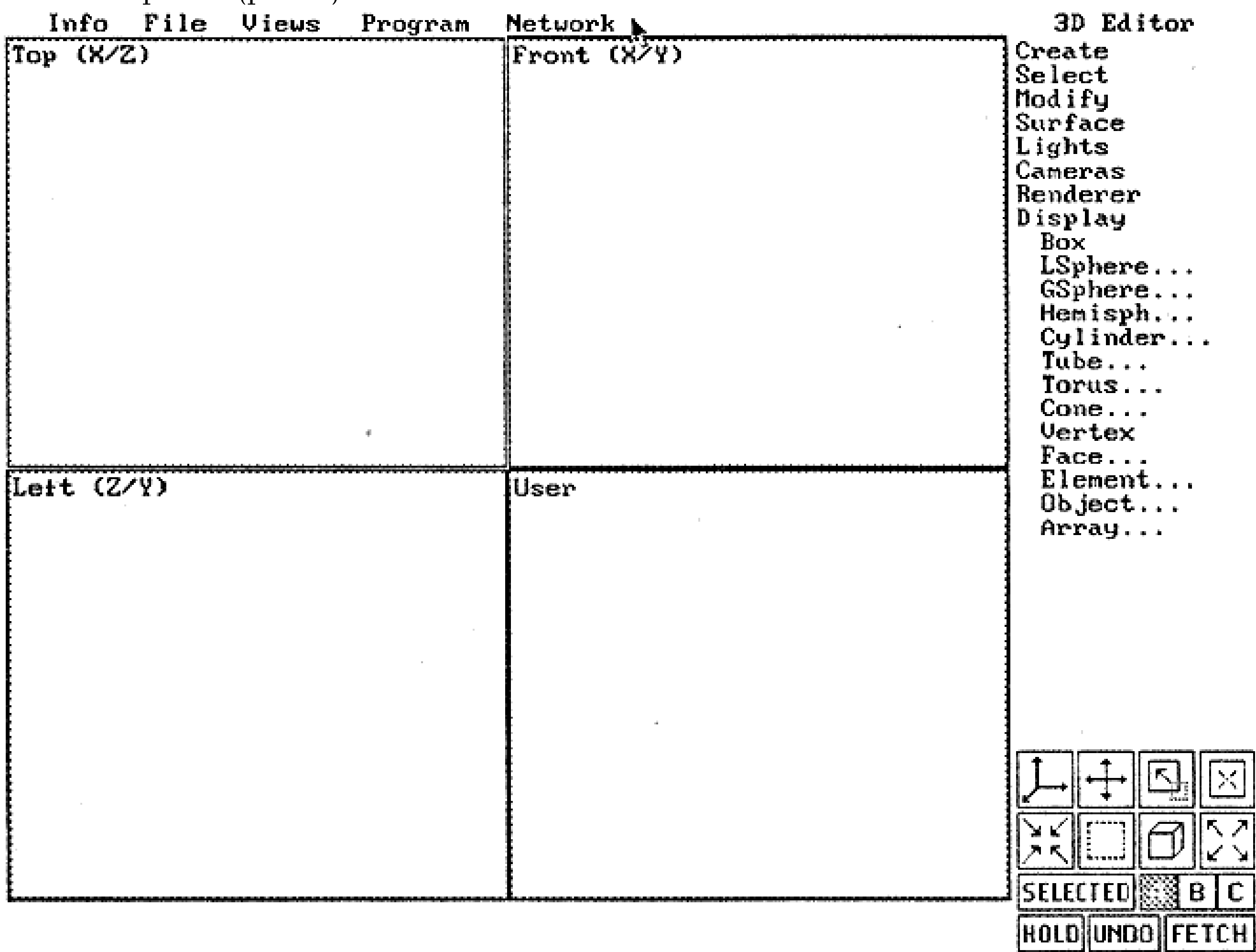


Рис. 1.

В верхней части экрана находится статусная строка, или меню (для того чтобы увидеть меню, достаточно установить курсор мыши в верхнюю строку). Ниже находятся окна (viewports), изображающие виды объектов с разных сторон (в 2dShaper'е окно только одно, так как все объекты плоские). При этом одно из окон (оно обведено белой рамкой) является активным. Для активизации окна достаточно установить в него курсор мыши и нажать Левую кнопку.

Две последние строки экрана содержат запрос пакета при выполнении очередной команды. Запрос текущей команды выдается белым цветом, а черным - запрос предыдущей команды или ее результат.

В верхнем правом углу находится название активного модуля; под ним расположено командное меню.

В нижнем правом углу находится панель инструментов (рис. 2).

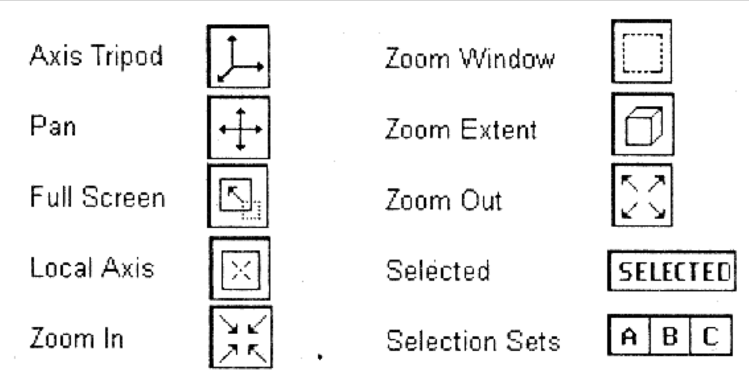


Рис. 2.

В 2dShaper'е статусная строка отображает координаты курсора, когда он находится в окне, например, [x:0.00 y:0.00].

Создание объектов

Существует несколько способов создания объектов в пакете 3D Studio. Самый простой заключается в использовании модуля 3d Editor для создания и модификации простейших объектов.

Второй способ - создание трехмерных объектов путем проноса (вытягивания) двумерной формы (образа, замкнутой линии) вдоль некоторой траектории (пути).

Создание отрезков и ломаных

Выберите из меню команд команду Create. В результате этого под меню команд появится список подкоманд для команды Create, сдвинутый вправо.

Выберите в этом списке команду Line (далее это действие будет обозначаться как Create/Line). Подведите курсор мыши к той точке, из которой вы хотите выпустить отрезок (при внесении курсора в окно он изменит свою форму и станет маленьким квадратом) и нажмите левую кнопку мыши. Затем установите курсор в точку, которую вы хотите соединить с выбранной ранее (при передвижении курсора за ним от первой точки будет тянуться прямолинейный отрезок) и нажмите еще раз левую кнопку мыши (появится отрезок, концы которого будут обозначены белыми крестиками). Установите курсор в следующую точку и нажмите кнопку мыши (появится прямолинейный отрезок, соединяющий предыдущую точку с только что выбранной). Таким образом можно построить ломаную линию, просто указывая ее последовательные вершины.

Для выхода из этого режима нажмите правую кнопку мыши (обычно правая кнопка всегда служит для отмены режима).

Редактирование построенной ломаной

Выберите в меню команду Modify/Vertex/Move и укажите вершину ломаной, которую вы хотите передвинуть (вершину можно передвинуть в любое место: для установки ее в текущую позицию надо нажать левую кнопку мыши, а для отмены перемещения - правую). Еще большие возможности дает команда Modify/Vertex/Adjust. Выберите какую-либо вершину ломаной и, не отпуская левую кнопку мыши, подвигайте ее.

В результате этих действий выходящие из вершины прямолинейные отрезки изогнутся, превратившись в кривые линии. Появятся желтый и красный векторы, приложенные к этой вершине (рис. 3). При дальнейших передвижениях мыши кривые будут изменять свою форму. Легко заметить, что желтый и красный векторы с началом в изменяемой вершине являются касательными к кривым, выходящим из этой вершины. Фактически каждый

сегмент ломаной (кривой) представляет собой кривую Эрмита, так как задаются не только концы сегмента, но и касательные векторы в концах. В начале процесса для каждого отрезка ломаной заданные векторы являются нулевыми.

Посредством изменения этих касательных векторов можно строить кривые линии с минимальным количеством вершин (ускоряя тем самым время рендеринга).

Например, окружность, создаваемая командой Create/Circle, хорошо описывается четырьмя точками с соответствующим образом подобранными касательными векторами.

Попробуйте теперь подвигать мышь, держа нажатой клавишу Ctrl, и вы увидите, что перемещается только вершина, а касательные векторы не изменяются.

Если вы будете держать нажатой клавишу Alt, то изменяться будет только желтый вектор. Тем самым вы получаете возможность индивидуальной настройки векторов.

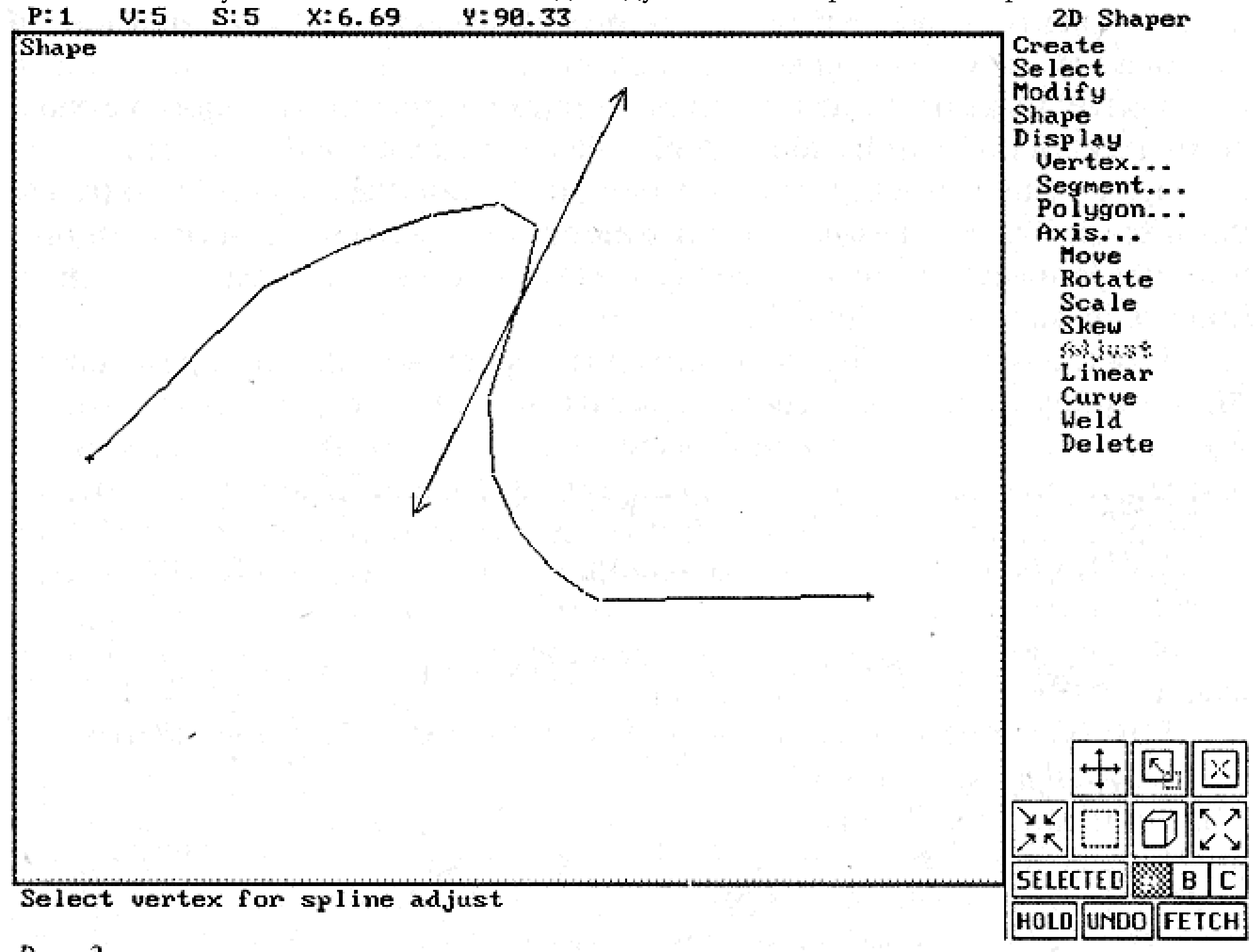


Рис. 3.

Очистите рабочую область, нажав клавишу N и выбрав Yes в появившемся диалоге. Теперь нажмите клавишу G (в окне появится сетка из точек (для управления шагом сетки используйте Views/ Drawing Aids из главного меню) и S (в верхнем правом углу экрана появится желтая буква S). Выберите Lines/Create и снова попробуйте построить ломаную. Вы увидите, что теперь курсор будет двигаться по узлам сетки скачками. Чтобы убрать передвижение только по сетке, еще раз нажмите клавишу S, а для убирания сетки - клавишу G.

Построение сложного объекта, состоящего из текста

Очистите рабочую область, выберите Create/Text/Font из меню команд и в диалоге выбора шрифта выберите BARREL.FNT. Затем, используя команду Enter, введите текст,

например 3dStudio. Для вывода текста на экран выберите команду Place, включите сетку и движение только по сетке (клавиши G и S), установите курсор в верхний левый угол сетки, нажмите и отпустите левую кнопку мыши. Выберите подходящий размер области, в которую вы хотите поместить текст, и еще раз нажмите левую кнопку мыши.

В результате в окне появится введенная надпись, изображенная выбранным шрифтом и растянутая до размера заданной области.

Получившаяся надпись скорее всего будет искажена - вытянутая или в ширину, или в высоту, она будет нарушать привычные пропорции букв. Для того, чтобы избежать этого, при выборе области следует держать нажатой клавишу Ctrl.

Вновь очистите область и выведите текст на экран, держа нажатой клавишу Ctrl, так, чтобы размеры текста (цифры в квадратных скобках в статусной строке) были 360.00 и 70.00 соответственно. Сохраните полученную надпись как форму (shape). Для этого выберите команду Shape/All (надпись станет желтой) и сохраните результат в файле EX1.SHP, нажав ^S, выбрав Shape Only и введя имя EX1.

Создание трехмерного объекта на основе построенной формы путем ее "протягивания" в глубину

Для создания объекта перейдите в 3dLofter, нажав клавишу F2. На экране появится уже не одно окно, а четыре: справа - большое окно Shape, где находится протягиваемая форма (сейчас оно должно быть пустым), а слева три маленьких окна: Top, Front и User, представляющие собой различные проекции трехмерного пространства.

Направления проектирования в окнах Top и Front являются фиксированными, а направление проектирования в окне User можно изменять при помощи пиктограммы Axis Tripod на панели инструментов или при помощи стрелок на клавиатуре.

В активном окне Top вы видите синий отрезок - путь, вдоль которого будет протягиваться надпись. Для получения формы из модуля 2dShaper выберите команду Shapes/Get/Shaper.

При этом на несколько секунд появится надпись Verifying shape validity. Это связано с ограничениями на протягиваемую форму - она должна состоять из одного или нескольких замкнутых многоугольников без самопересечений. После этого в окне Shape появится форма и белый крест - точка, к которой прикрепляется путь при протягивании.

Для установки этой точки в середину формы выберите команду Shapes/Center (более точная установка делается в 2dShaper при помощи команды Shape/Hook/Place). Чтобы и форма и путь были полностью видны во всех окнах, нажмите при помощи правой кнопки мыши (нажатие левой кнопки работает только для активного окна) на пиктограмму Zoom Extent.

Следующим шагом является коррекция пути (траектории, вдоль которой будет вытягиваться форма). По умолчанию путь представляет собой отрезок длиной 100 единиц.

Для дальнейшей работы желательно сократить длину пути, для чего выберите окно Top, нажмите клавишу W для раскрытия окна во весь экран и выберите команду Path/Move Vertex. Включите сетку и передвижение только по сетке.

Когда курсор мыши входит в окно, то он принимает форму квадратика с четырьмя стрелками, показывающими возможные направления для передвижения вершины.

Нажмите Tab несколько раз так, чтобы остались только две вертикальные стрелки. Опускайте верхнюю точку пути вниз до тех пор, пока длина пути (число в квадратных скобках в статусной строке) не станет равным 10.00, и еще раз нажмите клавишу W для возврата к нормальному расположению окон и сохраните результат в файле EX1.LFT.

Для создания трехмерного объекта выберите команду Objects/Make. В появившемся диалоге задайте имя объекта 3dStudio, выберите Smooth Length Off, Shape Detail Med и нажмите Create. После создания объекта перейдите в 3dEditor, нажав клавишу F3.

На экране вы увидите окна Top, Front, Left и User. Для начала при помощи пиктограммы Zoom Extent выровняйте надписи во всех окнах.

Создание источников света

Пакет 3D Studio поддерживает три типа источников света: фоновый (Ambient), определяющий глобальное освещение сцены, точечный (Omni) и направленный (Spot). Фоновый источник света может быть только один, он всегда существует и не имеет местоположения. Точечных источников может быть много. Каждый из них обеспечивает равномерное освещение объектов из заданной точки, но при этом объекты не могут отбрасывать тени. Направленный источник характеризуется не только своим местоположением, но также и точкой, на которую он направлен. Объекты, освещенные направленными источниками, могут отбрасывать тени.

Проектируя на освещаемый объект заданную картинку или фильм, направленные источники могут выступать и в роли проекторов.

Добавим в создаваемую сцену два точечных источника. Для получения равномерного освещения эти точечные источники должны быть расположены на достаточно большом расстоянии от освещаемого объекта; нажмите при помощи правой кнопки мыши два раза на пиктограмму Zoom Out (Zoom In и Zoom Out изменяют масштаб на 50%; для более тонкой регулировки держите нажатой клавишу Shift -при этом масштаб изменяется только на 10%). Для создания точечных ненаправленных источников света выберите из меню команду Lights/Omni/Create и активизируйте окно Top. Поместите один источник света в левый нижний угол, а другой - в правый верхний.

При создании источника света появляется диалоговое окно, запрашивающее имя источника и его параметры - интенсивность и цвет. Оставляя пока эти параметры без изменения (их можно изменить потом при помощи команды Lights/Omni/Adjust), активизируйте окно Front и выберите команду Move для перемещения источника света. Передвиньте левый источник вверх, а правый - вниз.

Создание камеры

Для расчета изображения необходимо задать окно, определяющее вид сцены. Для этой цели можно использовать любое из существующих окон, но удобнее всего создать камеру и назначить одно из окон как вид из камеры.

Камера задается двумя точками - своим местоположением (изображается большим синим кружком) и точкой, на которую она смотрит (маленький синий кружок).

Для создания камеры выберите в меню команду Cameras/Create и поставьте камеру в правый нижний угол окна Top.

После установки камеры за перемещающимся курсором мыши, определяющим вторую точку, потянется вектор, задающий направление обзора.

Укажите этим вектором на надпись и еще раз нажмите левую кнопку мыши. В появившемся диалоге выберите Create. Активизируйте окно User и нажмите клавишу C (название окна изменится на Camera01).

В окне Camera01 вы видите сцену такой, какой она выглядит из камеры. Используя команду Cameras/Move, настройте местоположение и точку обзора камеры для получения наилучшего вида на объект путем передвижения обеих точек.

Теперь, когда сцена уже построена, ее можно просчитать.

Расчет сцены

Выберите Renderer/Render View и укажите мышью на окно камеры (поместите курсор в это окно и нажмите левую кнопку мыши). Если перед этим это окно не было активным, то нажать кнопку нужно два раза.

В появившемся диалоге параметров рендеринга нажмите кнопку Render. Если вы правильно выполнили все шаги, то по окончании расчета на экране у вас должно получиться изображение серой надписи 3dStudio на черном фоне.

Выбор материалов

При желании можно выбрать другой, более красивый фон и сделать надпись не серой, а из какого-нибудь материала. Начнем с описания последнего.

При помощи команды Surface/Material/Choose выберите из предлагаемого списка материалов GOLD (DARK). После этого командой Assign/By Name вызовите список объектов (в данном примере он будет содержать всего один объект) и отметьте объект 3dStudio для назначения материала. Нажмите Ok и подтвердите, что вы действительно хотите назначить материал GOLD (DARK) объекту 3dStudio. Для выбора фона служит команда Renderer/Setup/Background. Нажмите при помощи мыши на поле справа от кнопки Bitmap. В появившемся диалоге нажмите на кнопку *.JPG и выберите файл CLDSMAP.JPG. Перед дальнейшими действиями сохраните сцену в файле EX1.3DS.

Выберите Renderer/Render View и окно Camera01. В диалоге параметров рендеринга выберите кнопку Backgound Rescale (попробуйте, что будет, если выбрать Tile). Для записи изображения в файл нажмите кнопку Disk. Для расчета нажмите Render и в диалоге выбора имени файла для сохранения результатов наберите EX1.

Добавим к построенной сцене еще один объект - мраморный стол, расположенный под надписью.

Для создания нового объекта перейдите в 3D Editor, выберите команду Create/Box и в окне Top постройте прямоугольник вокруг надписи. После того, как вы поставите вторую точку прямоугольника, он исчезнет и курсор опять примет форму перекрестия. При этом в нижней строке экрана появится надпись Click in viewport to define length of the box.

Теперь вам следует указать размер объекта в третьем измерении. Для этого в окне Left необходимо построить отрезок, задающий высоту объекта и после этого в появившемся диалоге задать имя объекта -Table.

Для того, чтобы поместить созданный объект под старым, выберите команду Modify/Object/Move и передвиньте объект "Table" в окне Left.

Теперь необходимо задать материал, из которого будет сделан брусок.

Используя команду Surface/Material/Choose, выберите материал MARBLE-TAN и с помощью команды Surface/Material/Assign/By Name назначьте выбранный материал объекту Table.

Если теперь осуществить рендеринг построенной сцены, то уже в самом его начале появится предупреждение - Object "Table" needs mapping coordinates - и запрос о продолжении рендеринга; и если все же выбрать продолжение, то в построенном изображении новый объект окажется черным.

Создание материалов. Свойства материалов

Для того, чтобы разобраться в причинах этого, перейдите в модуль Material Editor с помощью клавиши F5. После этого в верхней части экрана (рис. 4) появится меню этого модуля или статусная строка (в зависимости от положения курсора).

Ниже на экране расположены окна, в которые помещаются образцы материалов. Одно из этих окон является активным - оно обведено белой рамкой. При этом если вы используете 256-цветный режим, то цветное изображение будет находиться только в активном окне, во всех остальных находится черно-белое изображение.

Еще ниже находится поле, содержащее название материала. Вы можете легко изменить его, нажав на это поле с помощью мыши и введя новое название.

Далее располагается тип рендеринга для данного материала - Flat (плоский), Gouraud (Гуро), Phong (Фонга) и Metal (металлический). Правее находятся специальные опции - 2-Sided (материал является двусторонним) и Wire (соответствующий объект будет изображен в виде проволочного каркаса).

Плоский рендеринг представляет собой простейший вид рендеринга, когда освещенность каждой грани постоянна и не зависит от выбора точки на грани.

Закраска Гуро и Фонга полностью соответствует материалу, изложенному в главе, посвященной методам закрашивания.

Металлический рендеринг представляет собой разновидность закраски Фонга, отличающуюся обработкой бликов.

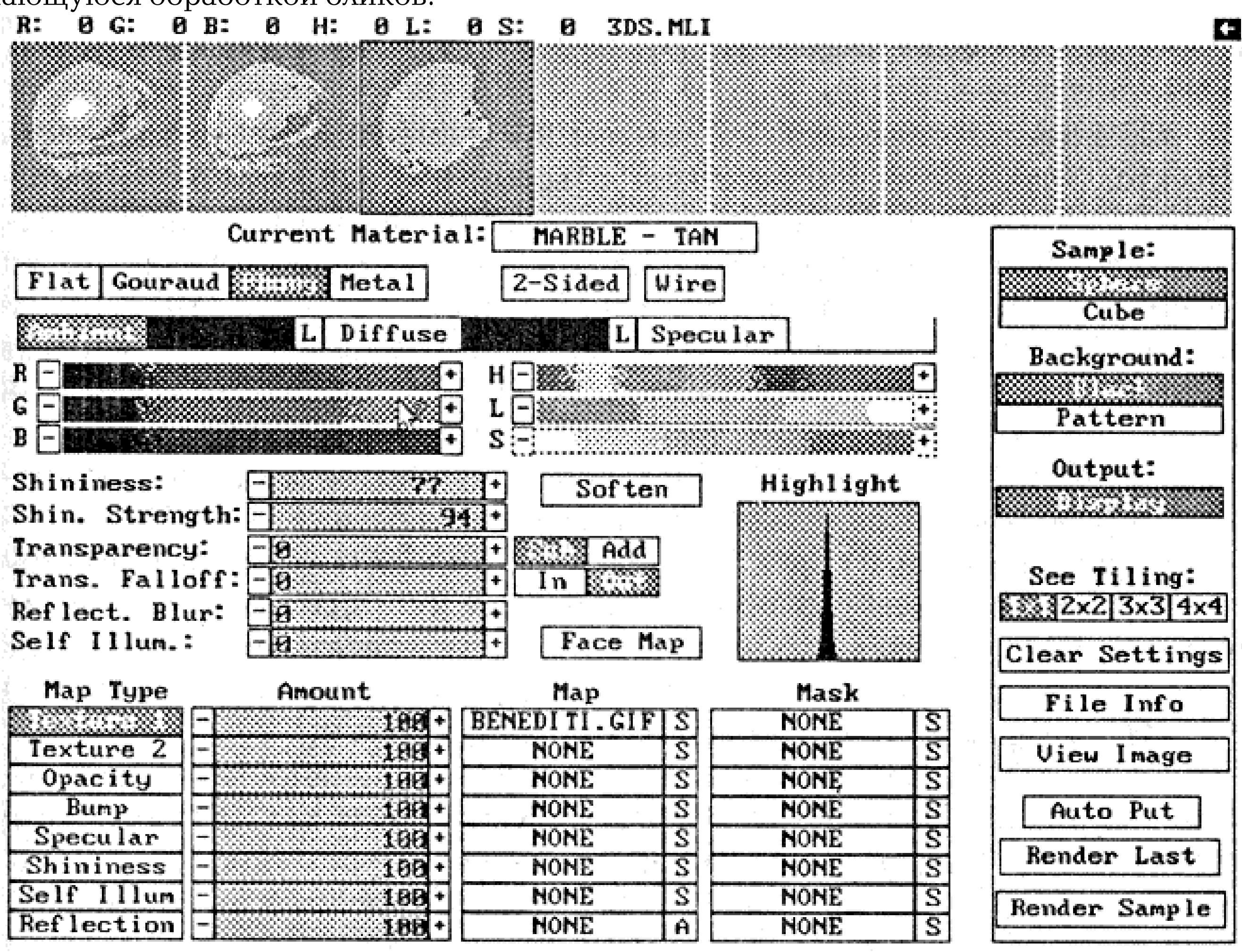


Рис. 4.

Понятие двусторонних и односторонних материалов связано с отсечением нелицевых граней, поэтому если необходимо этого избежать, то соответствующим граням назначается двусторонний материал.

Далее следует большая группа стандартных параметров материала, задаваемых при помощи движков. Среди них три цвета (при металлическом рендеринге - цветов только два) - фоновый (Ambient), диффузный (Diffuse) и зеркальный (Specular).

Идущие далее параметры Shininess и Shin. Strength определяют яркость и размер бликов. Результат их действия вы видите на графике в окне Highlight.

Параметр Shininess отвечает за резкость блика (чем больше его значение, тем меньше и контрастнее получается блик; значение 0 соответствует отсутствию бликов), а параметр Shin. Strength отвечает за яркость бликов. Фактически первый параметр соответствует степени p, а второй параметр - коэффициенту Ks в уравнении освещенности.

Для металлического рендеринга задаются только два цвета - фоновый и диффузный (где последний определяет и диффузное освещение, и цвет бликов). При этом параметр Shin. Strength отвечает за "металличность" материала.

Следующий параметр - Transparency - определяет прозрачность объекта. Значение 0 соответствует совершенно непрозрачному материалу, а значение 100 - совершенно прозрачному. Находящиеся справа кнопки Sub и Add определяют изменение цвета, проходящего через материал света - или из него вычитается цвет материала (Sub), или к нему добавляется цвет материала (Add). Обычно используется первый вариант, однако режим Add можно с успехом использовать и для создания объектов, выглядящих, например, как лучи света.

Параметр Transp. Falloff и кнопки In и Out определяют зависимость прозрачности от угла падения на поверхность.

Следующий параметр - Refl. Blur - определяет степень размытости отраженного изображения. Чем больше его значение, тем более размытым оказывается отражение. Однако этот параметр не оказывает никакого влияния на часто используемые плоские зеркала.

Параметр Self Ilium отвечает за свечение объекта.

Пакет 3D Studio предоставляет широкие возможности по использованию различных видов текстур, простейшими из которых являются проективные. Каждой карте (проективной текстуре) можно сопоставить еще одну карту, являющуюся маской (интенсивность точки маски определяет вклад карты в значение соответствующего параметра).

Стандартными типами текстур являются:

- цветовые карты (Texture 1, Texture 2), задающие цвет объекта в заданной точке; поддерживается до двух текстурных карт (для двух карт маска используется для смешения их между собой);
- карта прозрачности (Opacity), задающая прозрачность объекта при помощи интенсивности используемой карты;
- карта микрорельефа (Bump), использующая изменение интенсивности карты (ее градиент) для изменения вектора нормали к поверхности;
- карта цвета бликов (Specular), задающая цвет бликов;
- карта силы бликов (Shininess);
- карта самосвечения (Self Ilium), создающая эффект свечения объекта;
- карта отражения (Reflection), служащая для моделирования отражения; ее применение связано с тем методом, который используется пакетом для имитации отражений, при этом отраженное изображение трактуется просто как еще одна карта, накладываемая на поверхность.

Существует несколько типов карт отражения, начиная от просто заданного файла, например REFMAP.GIF, и до автоматических карт, которые сами проводят рендеринг сцены для создания карты отражения.

Справа от названия типа карты находится регулятор силы карты, название используемого файла пли NONE, если карта не используется, и кнопка S, служащая для задания параметров применения карты -сжатия (растяжения), угла поворота и других. Далее расположено ноле для имени файла, маскирующего используемую карту, и кнопка S для задания соответствующих параметров маски.

Для карты отражения вместо, стандартной кнопки параметров находится кнопка А, позволяющая создавать автоматическую карту отражения.

В правой части окна в колонку расположены кнопки, управляющие рендерингом образца материала. Первые две из них - Sphere и Cube - позволяют в качестве используемого образца выбрать сферу или куб.

Следующие две кнопки служат для выбора фона, на котором строится изображение образца. Можно задавать как черный фон (Black), так и шаблон из цветных клеток (Pattern), что особенно удобно для работы с прозрачными материалами.

Кнопки 1 x 1, 2 x 2, 3 x 3 и 4 x 4 задают повторение карт на образце.

Кнопка Render Sample осуществляет рендеринг образца для выбранных параметров, так как простое их изменение не вызывает перерасчета изображения образца до нажатия этой кнопки.

С помощью команды меню Material/Get Material загрузите в материал MARBLE - TAN. В первом окне вы увидите образец, сделанный из этого материала, а в остальных полях - параметры материала.

При этом видно, что выбранный материал использует в качестве текстурной карты файл BENEDITI.GIF. Активизируйте второе окно образцов и загрузите туда материал GOLD (DARK). Вы легко можете убедиться, что этот материал использует файл REFMAP.GIF в качестве карты отражения.

Для использования любой проективной текстуры (кроме карты отражения) необходимо, чтобы для объекта, использующего данный материал, был задан способ проектирования используемой карты на поверхность объекта. Это отображение называется mapping; и именно то, что оно не задано в нашей сцене ни для одного объекта, приводит к тому, что материал MARBLE - TAN не может быть использован (изображение соответствующего объекта получилось черным).

Пакет 3D Studio поддерживает три различных типа проектирования карты на объекты - плоское (параллельное), цилиндрическое и сферическое проектирование.

Для установки нужного типа проектирования вернитесь в 3D Editor и при помощи команды Surface/Mapping/Type/Planar выберите плоское проектирование.

В результате вы увидите прямоугольник с зеленой и желтыми сторонами (map icon). Он определяет, каким способом карта будет накладываться на объект. Вы можете рассматривать его просто как образ карты. При использовании плоского проектирования желательно так ориентировать этот прямоугольник, чтобы нормаль к нему (направление проектирования) не была параллельна касательной плоскости ни в одной точке объекта.

Для введенного нами объекта это условие означает, что прямоугольник не должен быть параллелен ни одной из сторон параллелепипеда. Чтобы добиться этого, следует повернуть этот прямоугольник в двух окнах, используя команду Surface/Mapping/Adjust/Rotate. Поскольку прямоугольник представляет собой образ накладываемой карты, то в случае, если этот образ не накрывает весь объект, он будет циклически повторяться со сдвигом.

При этом край изображения скорее всего будет бросаться в глаза. Чтобы этого избежать, образ карты следует промасштабировать так, чтобы он полностью накрывал объект. Для этого служит команда Surface/Mapping/Adjust/Scale. После того, как образ будет настроен, необходимо назначить его объекту при помощи команды Surface/ Mapping/Apply Obj.

Для ее использования выделите объект Table, например при помощи команды Select/Object/By Name, выберите команду Surface/Mapping/Apply Obj и в появившемся диалоге выберите Yes. Сохраните построенную сцену в файле и осуществите ее рендеринг.

При этом вы должны увидеть нормальное изображение сцены с мраморным блоком и надписью.

Добавим к этой сцене отражение - и блок будет отражать находящуюся над ним надпись.

Поскольку отражение является одним из атрибутов материала, то необходимо создать новый материал (назовем его MARBLE MIRROR) с назначенной картой отражения.

Для этого вернитесь в модуль Material Editor, активизируйте окно, соответствующее материалу MARBLE - TAN, и измените его название на MARBLE MIRROR. Для установки карты отражения выберите мышью кнопку A в строке карты отражения. После этого в поле имени файла появится слово AUTOMATIC, означающее, что выбрана автоматически генерируемая карта отражения. Для настройки ее параметров с помощью мыши выберите это поле и в появившемся диалоге выберите тип карты - Flat Mirror.

Это самый простой вариант карты отражения; но он может назначаться только тем граням объекта, которые лежат в одной плоскости.

С помощью команд меню Put Material и Save Library сохраните новый материал в библиотеке.

Теперь необходимо назначить созданный материал верхним граням блока (поскольку все объекты состоят из треугольных граней, то наш брусок имеет две верхние грани).

Перейдите в модуль 3D Editor и выделите эти две грани при помощи команды Select/Face/Quad. При этом вы выделяете в одном из окон - Front или Left - прямоугольную область, и все грани, попавшие в этот прямоугольник, выделяются.

В рассматриваемом случае прямоугольник можно построить в окне Front. При этом ребра, соответствующие выделенным граням, окрашиваются красным.

Для назначения материала сначала выберите сам материал, затем нажмите кнопку SELECTED; для того чтобы следующая операция относилась к выделенным граням, воспользуйтесь командой Surface/Material/Assing/Face и укажите мышью в активное окно.

Анимация

Для придания готовой сцене динамики служит модуль Keyframer. Он позволяет легко и естественно добавлять в сцену движение, основываясь на понятии ключевых кадров - небольшого количества основных кадров, в которых задано преобразование объектов. При этом на все остальные кадры эти преобразования интерполируются, так что для создания простейшей анимации достаточно задать преобразование в последнем кадре.

Добавим к созданной ранее сцене вращение надписи вокруг вертикальной оси. Для этого необходимо перейти в Keyframer нажатием клавиши F4. Содержимое экрана (рис. 5) в этом модуле мало отличается от содержимого экрана в других модулях, но в правом нижнем углу появились дополнительные иконки для переходов между кадрами.

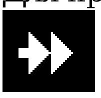
Рис. 5.

Для задания простого вращения достаточно задать поворот на 360° в последнем кадре. Для перехода к нему нажмите на иконку



При этом изображение объектов будет выведено не белым, а черным цветом, показывающим, что задается не геометрия сцены, а ее изменения во времени. С помощью команды Object/Rotate поверните надпись в окне Front на 360°. Для задания правильной оси, вокруг которой будет осуществляться поворот, необходимо два раза нажать на клавишу Tab, чтобы в верхней строке появилась надпись Axis: Y.

Для просмотра получившейся анимации нажмите на иконку



и вы увидите вращение надписи.

При этом вращение будет плавным за исключением самого первого кадра, где движение будет как бы замирать на один кадр. Это связано с тем, что в построенной последовательности кадров первый и последний кадры совпадают и поэтому плавность движения нарушается.

Чтобы при расчете сцены это было незаметно, следует задать диапазон кадров, которые вы будете рассчитывать. Для рендеринга сцены выберите, как и ранее, команду Render/Render View и окно камеры. Но в отличие от предыдущего случая карточке параметров рендеринга следует выбрать поле Range (для рендеринга только заданного набора Кадров). Для того, чтобы анимация была записана в файл, следует также выбрать поле Disk.

Создание объектов вращения

С помощью модуля 3d Lofter можно очень легко создавать тела вращения. Проиллюстрируем это на примере создания чашки с блюдцем.

Для этого в модуле 2d Shaper постройте изображение, представленное на рис. 6. Для создания по этому скелетному изображению контура конструируемых объектов выберите для каждой ломаной команду Create/Outline, выберите линию и задайте небольшое расстояние.

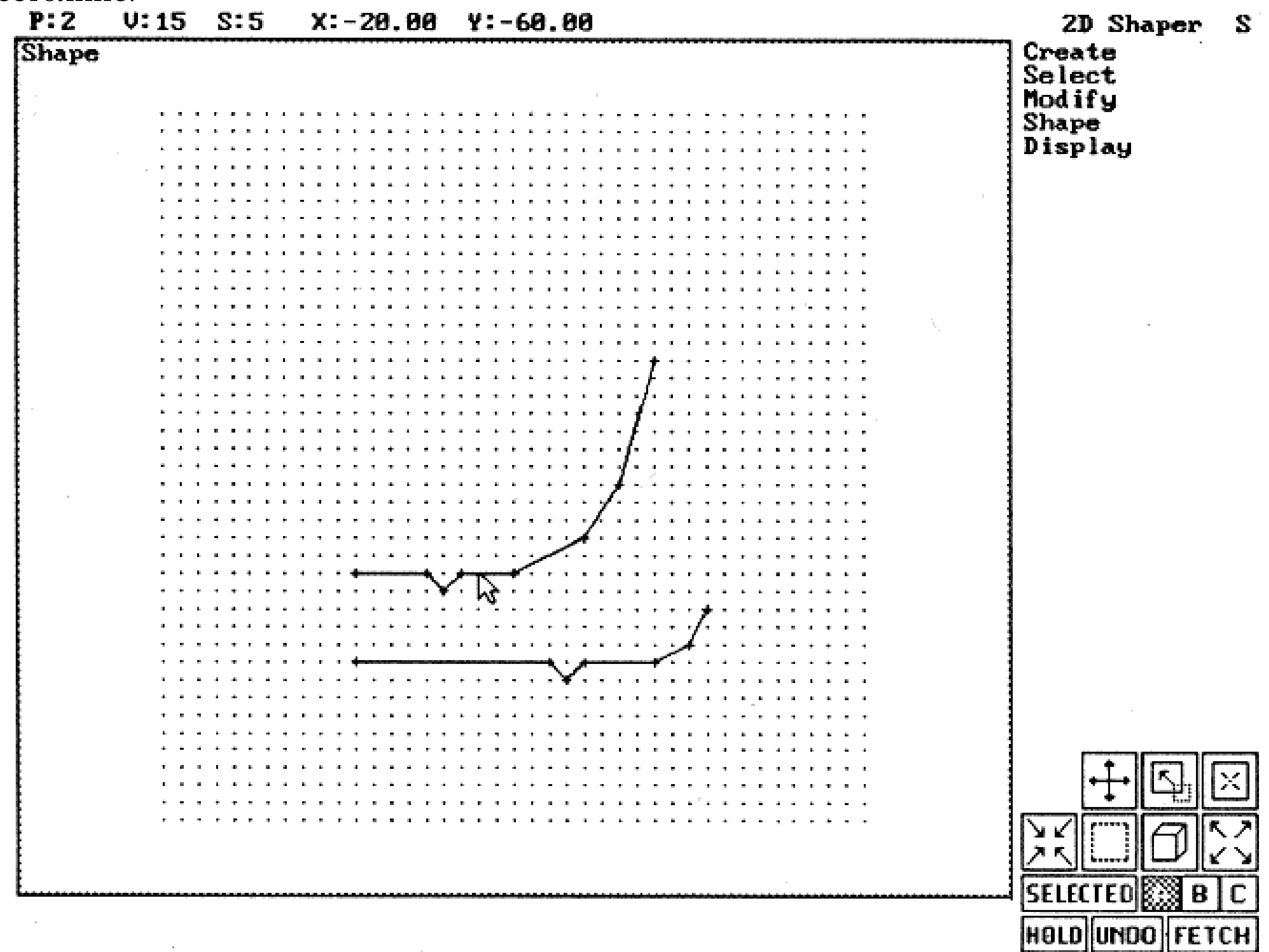


Рис. 6.

Следующим шагом будет сглаживание полученного контура при помощи команды Modify/Vertex/Adjust для построения изображения, представленного на рис. 7.

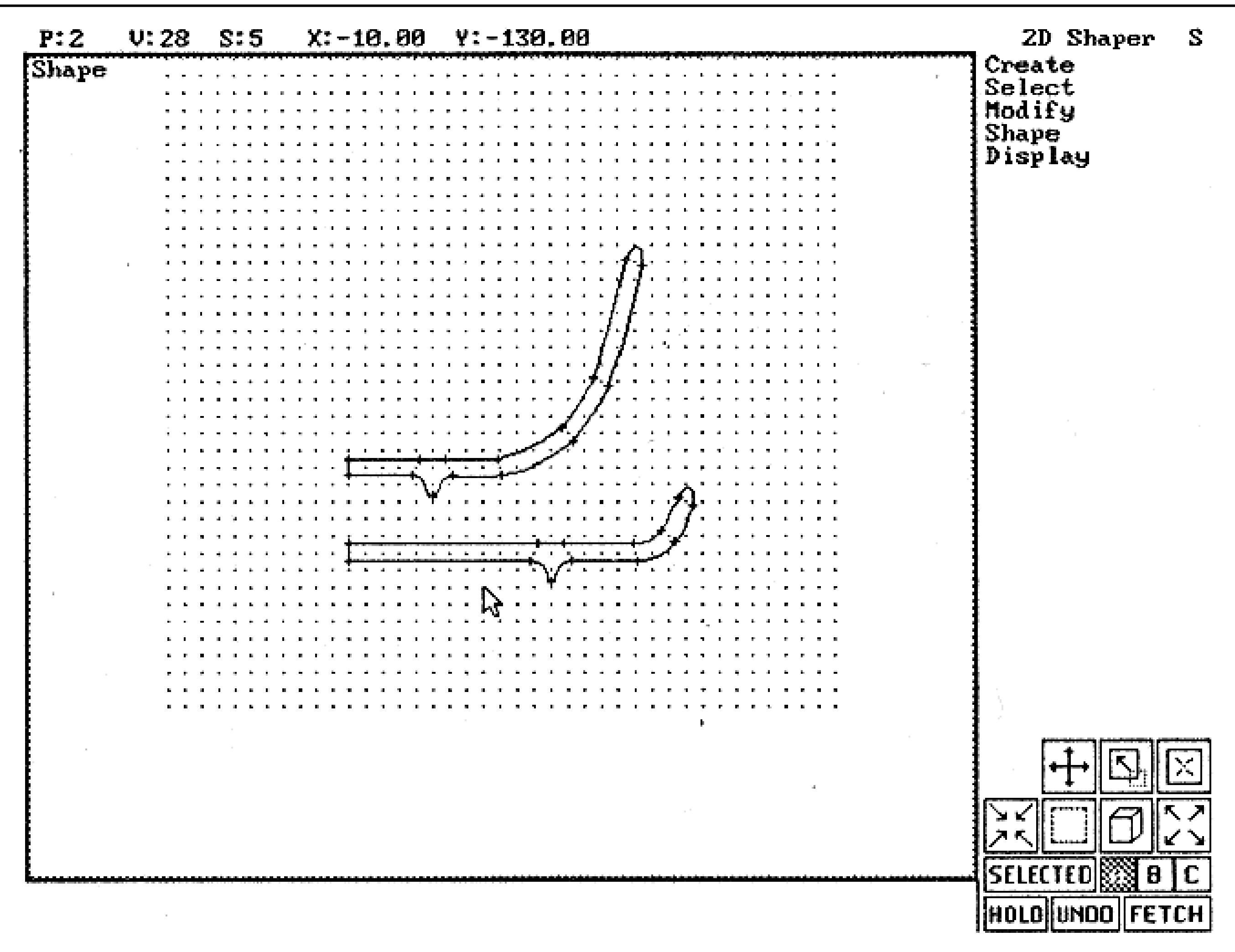


Рис. 7.

Для того, чтобы построить поверхность вращения, перейдите в 3d Lofter и постройте кольцевой путь при помощи команды Path/SurfRev. Следующим шагом будет получение формы из модуля 2d Shaper при помощи команды Shapes/Get/Shaper и выравнивание ее на пути при помощи команды Shapes/Center. Для построения объекта воспользуйтесь командой Objects/Make.

ЛИТЕРАТУРА

Роджерс Д., Адамс Дж. Математические основы машинной графики. -Машиностроение, 1980.
Гилой В. Интерактивная машинная графика. - Мир, 1982.
Фокс Ф., Пратт М. Вычислительная геометрия. Применение в проектировании и на производстве. - Мир, 1982.
Ньюмен У., Спрулл Р. Основы интерактивной графики. - Мир, 1985.
Фоли Дж., ван Дэм Ф. Основы интерактивной машинной графики. - Мир, 1985. Математика и САПР. В 2-х книгах. - Мир, 1988.
Barsky B. Computer graphics and geometric modeling using Beta-splines. — Springer Verlag, 1988.
Павлидис У. Алгоритмы машинной графики и обработка изображений. - Радио и связь, 1988.
Glassner A., editor. An introduction to ray tracing. - Academic Press, 1989.
Farin G. Curves and surfaces for computer aided geometric design. A practical guide. -Academic Press, 1990.
Stevens R.T. Fractal programming and ray tracing with C++. - M&T Books, 1990. Upstill S. The RenderMan companion. A programmer's guide to realistic computer graphics. - Addison-Wesley, 1990.
Foley D.J., van Dam A., Feiner S.K., Hughes J.F. Computer graphics. Principles and practice. - Addison-Wesley, 1991.
Hall R. Illumination and color in computer generated imagenary. - 1991.
Аммерал Л. Машинная графика на языке С. В 4-х книгах. - Сол Систем, 1992.
Иванов В. П., Батраков А.С. Трехмерная компьютерная графика. - Москва, Радио и связь, 1994.
Лорен Хейни. Построение изображений методом слежения луча. - Москва, 1994.
Уилтон Р. Видеосистемы персональных компьютеров IBM PC и PS/2. Руководство по программированию. - Москва, Радио и связь, 1994.
Шикин Е. В., Боресков А. В., Зайцев А. А. Начала компьютерной графики. -Москва, ДИАЛОГ-МИФИ, 1993.

Авторы считают целесообразным обратить внимание на ряд журналов, доступных российскому читателю, которые с известной регулярностью публикуют статьи, посвященные различным вопросам компьютерной графики. Это, в частности, "Компьютер-Пресс", "Мир ПК", "Монитор" и "Монитор-Аспект".

СОДЕРЖАНИЕ